MASTER IN COMPUTER SCIENCE

Design and implementation of a graphical editor for concurrent systems

de Menten, Gaëtan

Award date:
2004

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'Informatique
Année académique 2003-2004

Design and implementation of a graphical editor for concurrent systems

Gaëtan de Menten

Mémoire présenté en vue de l'obtention du grade de Maître en Informatique.

Résumé

Le "model checking" est une famille de méthodes dont le but est de vérifier automatiquement que des programmes (ou, plus généralement, des systèmes digitaux) satisfont à leur spécification. Dans ce contexte, la spécification d'un système est composée de plusieurs propriétés exprimées en utilisant une logique temporelle. Les systèmes coopératifs sont de bons candidats pour faire l'objet de vérifications automatiques. Une possibilité de visualisation des systèmes à vérifier est un attrait non négligeable pour un outil de "model checking". Cependant, il est encore plus attrayant pour ces outils de disposer d'un éditeur graphique permettant de créer et modifier le modèle de ces systèmes. Dans ce travail, nous présenterons un éditeur graphique pour systèmes coopératifs, et son implémentation.

Mots clefs : model checking, logique temporelle, visualisation de systèmes coopératifs, éditeur graphique

Abstract

The model checking is a family of methods aimed at automatically verifying that programs (and digital systems in general) satisfy their specification. In this context the specification of a system is made out of several properties expressed using temporal logics. Concurrent systems are good candidates for automatic verifications. Model checking tools can greatly benefit from offering a visualization of the systems they verify. Better still is to have a graphical editor to create and edit the model of these systems. In this work, we shall present a graphical editor for concurrent systems and its implementation.

Keywords : model checking, temporal logic, visualization of concurrent systems, graphical editor

Acknowledgements

First, I would like to thank my promoter, Jean-Marie Jacquet, for his patience, his understanding, and his advice.

My thanks also to the ParaDiSe laboratory in Brno, Czech Republic, where my foreign practical training took place, and its chief of staff, Lubos Brim. My special gratitude goes to David Safranek, who enlightened my work there.

Thanks also to my parents and close friends for their continuous moral support during the writing of this thesis. Further thanks go to my father, Benjamin Gigot and Patrick Aussems for the time they spent to proofread this paper and for the numerous and insightful suggestions they offered.

Finally, I am grateful to the whole free software community for giving me the opportunity to conceive and write this thesis using free and quality software.

Contents

Introduction

Since the birth of computers and of programming, the complexity and the size of programs has steadily increased. As a consequence of this, problems in design arose and increased, as well as difficulties in verifying the correctness of programs. This increase was proportional if not exponential.

And these difficulties are not limited to software programs: in fact, verifications of digital systems, whether software or hardware, started to experience the same kinds of problems. In concurrent systems the problems are much worse and there is nowadays a tendency towards building highly-concurrent systems: today most programs (of a certain size) are either explicitly built, or can be considered, as a system with several concurrent processes or components.

Further to the above, many systems cannot afford errors. These can be very expensive, especially in systems whose access is denied or difficult and/or expensive after they leave the hands of their designers. One does not need to go far to find examples of potential expensive errors: controllers for vital systems such as nuclear plants or airline traffic, medical equipment, rocket guidance systems (for ex. one Ariane launch failed due to a software glitch), space exploration vehicles (Mars Exploration Rover), satellites, defective hardware that needs to be recalled on a large scale (cars, for example), etc.

What was said above lead to search for ways to automatically prove the correctness of concurrent systems. Trying to automatically validate such programs is not a new topic, but the techniques developed so far were either impractical to automate (for example, invariants techniques) or limited to a specific class of programs (Hoare logic is only suited for programs of which output is a direct function of their initial input).

That induced to search for other ways to validate system specifications. One of the slightly more recent ways that were proposed was to use *temporal logics* (which is an extension of first-order logic) to describe what are the possible "good behaviors" of systems in terms of the sequence of states they can be in. And subsequently *model checking techniques* appeared in order to verify that the system actually conforms to this "good behavior description".

Verification tools (providing *model checking techniques*) traditionally work on models of systems rather than directly on the systems. Using verification tools has traditionally been quite arduous, one part of the problem being to feed them with a *model* for the system to be verified. Having a graphical user interface for those tools can already ease the approach. In addition to this, visualizing a graphical

representation of the model of a system greatly helps most people to understand it. Being able to graphically create or edit such models is therefore the logical next step to further facilitate the use of verification tools. This naturally lead to graphical editors appearing in the domain.

In this work, we shall present "Paxion", which is our graphical editor for concurrent systems[1]. It was specifically designed to work as a "front-end"[2] for DiVinE (a Distributed Verification Environment developed by the ParaDiSe Laboratory[3]).

In the first chapter of this paper, we will start by introducing the reader to some general concepts about model checking. We will explain in more details only one of its numerous possible approaches : LTL Model Checking, which is the method currently available in DiVinE. Also, as a general rule, in that first chapter, we will only elaborate on the concepts which we will need later in our discussion.

Our second chapter will go further into details in two directions : we will first discuss the possible visualizations of (complex) concurrent systems, and thereafter will examine what to expect from a graphical editor for concurrent systems and what exists in that domain.

The aim of these first two chapters is essentially to provide the reader with the concepts and enough information to put into perspective our own editor which is presented in the last two chapters.

In the third chapter, we present our editor in detail including its features, an example of how to use it to model a system, its architecture, and finally some details about some of the interesting parts of its implementation.

The fourth and last chapter will be devoted to a critical analysis and possible improvements of what we have done.

[1] the interested reader may find packages for the latest releases (including source code and automatically generated documentation) at `http://www.info.fundp.ac.be/~gdemente/paxion`

[2] a graphical user interface on top of a "back-end", which does the actual work

[3] http://www.fi.muni.cz/paradise/

Chapter 1

Model Checking

In this chapter, we will introduce the reader to model checking concepts. Model checking could be defined as the action of verifying properties of a system. A system can be of many different types such as a software program, a digital circuit, a protocol, etc.

Since one can hardly work directly on a concrete system, the first step in model checking is, as the name suggests, to build a model of that system. The properties one wants the system to have will be checked on that model and not on the system itself. That leaves us with three problems: how to model the system, how to specify the properties we want it to have and how to check that the model actually verifies those properties.

In this chapter we will begin with a quick reminder on Finite State Machines and progressively extend these to obtain a model complete enough to model concurrent systems. Then we will explain how one can specify the desired properties of such a system using temporal logics. Finally we will briefly introduce the reader to algorithms that can be used to perform the model checking itself (to verify that the system has the desired properties). This chapter is mainly inspired from [1], [8] and the course *"Preuve automatique et preuves de programmes"* given by Prof. Pierre-Yves Schobbens with insight from many other articles. We will also sometimes refer to DiVinE for which the main source of information was [4].

1.1 Model

There are many different formalisms and languages one can use to build a model of a system. The type of model chosen will depend mainly on what one wants to do with the model. Some formalisms are very specific for certain kinds of usages, while some others are quite general. Some examples of formalisms used in computer science include the Unified Modeling Language (UML) which was designed to model the different aspects of software systems, the Entity-Relationship model which concentrates on modeling data in information systems and databases, the Flowchart Model to represent algorithms, etc.

Even in the more particular field of model checking, many different formalisms

are used, such as Finite State Automata, Petri nets, recursive functions, lambda calculus, to name a few. But for the rest of this chapter, we will concentrate only on Finite State Automata, since they (and their variants) are probably the most common model.

1.1.1 Finite State Machines

A Finite State Machine (FSM), often also called Finite State Automaton (FSA) or simply Automaton, is an abstract machine (that is, a behavioral model) of a system.

It consists mainly of a (finite) set of states and a set of transitions between those states. The machine starts in its initial state and then evolves depending on the input fed to the machine. The input is read one symbol at a time and for each symbol of the input[1] a transition is triggered, so that the machine goes from one state to another (possibly the same). The alphabet of the input must be fixed and in each state, there should be one transition defined for each possible letter (symbol) of the alphabet of the input.

An automaton also has a set of accepting (or "terminal") states. When an automaton reaches one of these states, it stops.

Finite state machines can be divided in two categories depending on whether they produce output or not.

In the category of machines which do not generate output [2], we further differentiate machines which are deterministic from those which are not. In a Deterministic Finite Automaton (DFA), there is one, and only one, valid transition for each (state, input) pair, whereas in a Non-deterministic Finite Automaton (NFA) there can be several. As a side note, deterministic finite state machines are sometimes used in language theory to define regular languages (a word is part of the language if, and only if, it is accepted by the DFA—this is Kleene's theorem).

Note that it is possible to (automatically) translate a nondeterministic finite automaton into a deterministic one but the deterministic automaton will be bigger (in the worst case, exponentially) than the nondeterministic one.

In the category of machines generating output [3], there are two well known types of machines: *Mealy machines*, which associate some output to each of their transitions (the output is produced whenever the transition is "executed"), and *Moore machines* which have their output tied to their states (the output is produced whenever the state is reached).

[1] in this context, a symbol should not be exclusively understood in its literal sense but rather as an atomic piece of input or single input "event"

[2] usually called *acceptors* or *recognizers* in language theory because they either accept/recognize the input or not.

[3] usually called *transducers* in language theory because they take some text in input and generate some other text as output

Formal definitions

A deterministic finite state machine can be defined more formally as a 5-tuple composed of:

- a set of states (S),
- a set of input symbols (Σ),
- a transition function ($\delta : S \times \Sigma \rightarrow S$),
- an initial state ($s_0 \in S$),
- a set of accepting states ($A \subseteq S$).

For the nondeterministic version, the only difference is that the transition function returns a set of states instead of a single state. This can be written: $\delta : S \times \Sigma \rightarrow 2^S$

Example

Let's suppose we want to build a system (software or hardware) which recognizes one word. Such an automaton accepts the word if it receives the letters of the word in the correct order. We also want the automaton to accept the word even if it receives wrong letters before the word.

To keep the example simple, we will use the word "bad". This word has the advantage of being short and using letters only from the beginning of the alphabet, so we can pretend that the alphabet has only four letters: a, b, c and d. This automaton would have four states: the first one when no correct letter has been received yet, the second after a "B" has been received, the third after "BA", and the fourth, and accepting state, after "BAD" has been received.

There are eight transitions for that automaton including one transition for each of the "correct" letters if they are received in the correct state. All the other transitions are for "resetting" the automaton to the appropriate state if a wrong letter is received. This is where our reduced alphabet helps: it reduces dramatically the possibilities of wrong letters the automaton can receive, and as such reduces the number of these "reset" transitions. Even with that reduced alphabet, the task to list them all explicitly would be quite cumbersome. So, instead of listing them, we will rather skip directly to the next point which should clarify things.

Graphical representation

The previous example, although simple is not easy to grasp with a textual description like the one given. For most people it is easier to "read" a "visual description" of a model than a textual one. Like most formalisms for modeling systems, finite state machines have a graphical representation.

The usual representation for a finite state machine is a state diagram. The latter is a directed graph (i.e. a graph with vertices and edges joining them, the later

having a direction). The states, which are the vertices of the graph, are usually represented by circles and the transitions by labeled arrows. The initial state is usually differentiated from other states by having an arrow without origin pointing at it while the accepting states are differentiated by having a second (smaller) circle inside them. The label of the arrows can represent many different things, depending on which type, variant or extension of FSM is used.

The graphical representation for the example above can be seen on figure 1.1.

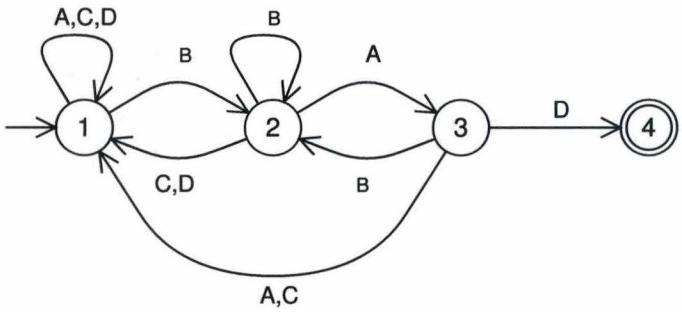

Figure 1.1: An automaton recognizing the word 'bad'

We will further discuss the topic of visualizing state machines, their extensions and other models in the second chapter.

1.1.2 Finite State Machines in Model Checking

There are many different variants and improvements of finite automata. Let's list a few of them: automata with empty transitions (ε-NFA) which can have "spontaneous" transitions (transitions on "empty" input), Generalized Nondeterministic Finite State machine (GNFA) which have their transitions labeled with regular expressions instead of single letters, pushdown automata (PDA) which have a stack to store data, automata on infinite words, etc.

We will not use the majority of these, since, in the above list, we will only develop automata on infinite words which are of special interest for model checking. In the next sections we will introduce several small extensions in order to be able to model and check concurrent systems but these are not part of the above list.

But before continuing we need to introduce a few new notions.

A *run* of an automaton is one possible execution path of a non-deterministic automaton.

A correct (accepted) run (behavior) of an automaton is one which ends in an accepting state. An example of a bad (ie not accepted behavior) is when it loops forever without ever reaching an accepting state.

A *reachable* state is simply a state which can be reached directly or indirectly from the initial state of a process. Alternatively, we can rephrase that with a recursive definition: a state is *reachable* if there is a transition leading to that state from either the initial state or another reachable state.

1.1.3 Variables, effects and guards

Let's try to model a system slightly bigger than our "BAD" example seen previously. The figure 1.2 represents one game in a tennis match. The transitions moving to the right (or up) represent a point won by the player (or team) A, and the transitions moving down (or left) a point won by player B. The 16 states in the top left are labeled by the scores of the players (score of player B/score of player A).

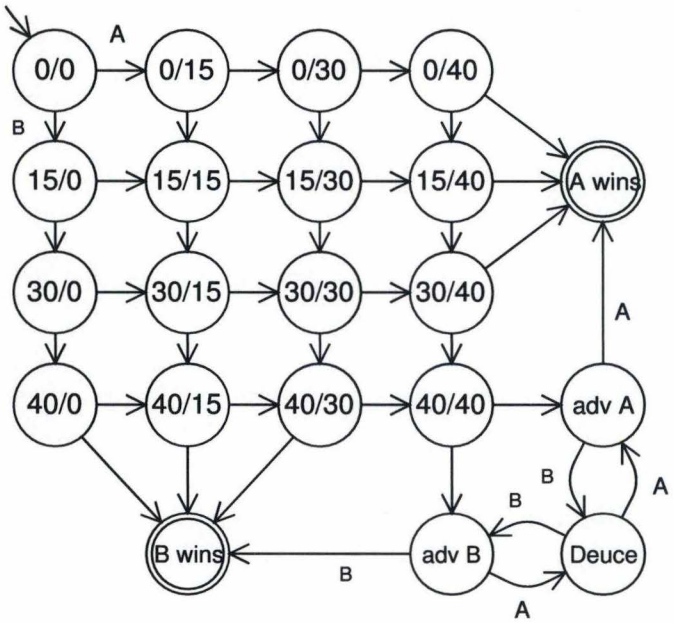

Figure 1.2: A model of a tennis game

While not incredibly complex this example counts already 21 states and takes some time to model by hand. And what if we want to model a whole set[4] of a tennis match and not only a single game? Even if we simplify the model and only count up to 6, we would have to duplicate the whole automata of figure 1.2 for each of the 36 score possibilities (0/1, 0/2, ..., 0/6, 1/1, 1/2, ..., 6/6), and that would take us already to 756 states.

By now it should be clear that it's not viable to model complex systems without some way to merge some of them together. The most obvious way in our example is to start counting points won instead of the score [in the tennis way] and then introduce variables. By introducing two variables, a and b, representing the scores of the two players, we can merge the 16 top-left states into a single state as shown on figure 1.3. Along with the introduction of variables comes another (obviously needed) concept: transitions can modify variables. From now on, we will call such a modification an "effect".

We now need to connect this newly merged state to the rest of the automaton. The problem we face is that we should only be able to leave that merged state if at least one of the two players has won four points. To resolve the problem, we

[4]In tennis, a set ends when a player has won 6 games and leads by at least 2 games

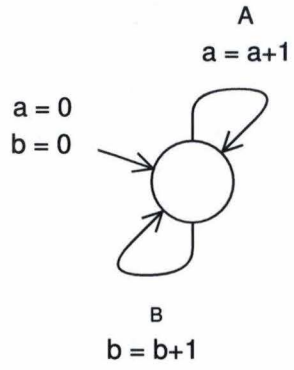

Figure 1.3: A (partial) model of a tennis game with variables

just introduce a condition—called a *guard*—which needs to be fulfilled before the transitions leaving the state can be executed.

Graphical representations for the *guards* vary slightly from one publication to another. In this chapter we will adopt label transitions by 'if', followed by the condition of the guard. Note that we use the C/C++/Java languages conventions for conditional statements and assignations ; so, 'testing for equality' is done with a "==" sign. As far as the *effect* is concerned, it is represented as one more label on the transition. Labels appear in this order: guard, input, effect. For our 'tennis game' example, the full graph with variables and guards can be seen on figure 1.4.

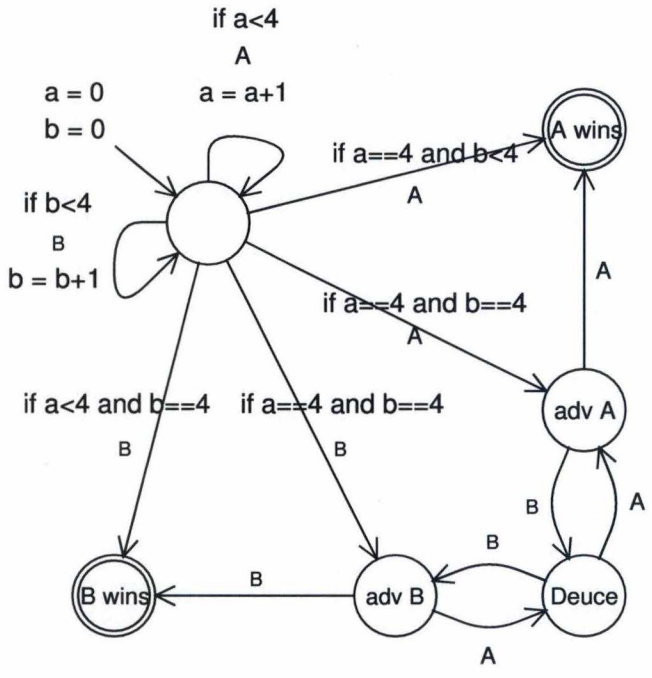

Figure 1.4: A model of a tennis game with variables

Variables in automata are obviously also very handy when modeling software programs since they map directly to the variables used in the programming language used to code the software. In that context one can see the states and transitions

of an automaton as representing the control flow of a program and the variables of that automaton as the data.

The process of transforming a model without variables to one with variables is called "folding" (and conversely transforming a model with variables to the corresponding one without variables is called "unfolding"). It is worth noting that many model checkers work on unfolded models. Therefore, what will usually happen is the opposite of what we have done above for the sake of the explanation: the system is first modeled with variables and then (automatically) unfolded before being "model checked".

As we already have hinted before, the unfolded automaton has many more states than the folded one. But we can rephrase that statement more precisely as follows : each "folded" state in the folded automaton is unfolded to a number of distinct states equal to the multiplication of the number of possible values for each variable it uses. In our tennis example our single "folded" state was expanded to 16 states because each of the two variables it uses can have four different values.

An interesting fact to highlight is that in the unfolded model there are no guards. There is no need for them since in each state we explicitly know the value of each variable. So it is important for variables to be bounded in the model (but this is rarely a problem since they are bounded anyway in most programming languages and hardware systems).

1.1.4 Concurrency

Let's assume we want to model a large system which is made of smaller, mostly independent, modules (this is a common practice in software engineering). Whether they are really only modules of a single program running on one machine or are part of a distributed program located in different physical places is of little importance in this paper.

Let's suppose we have already modeled those smaller modules (in the sequel, we will call these modules *processes*). How do we build a model for the whole system?

The model of the whole system is a product of the individual processes. One state of the global system represents in fact one of the possible combinations of states of its processes (and the content of the variables of such system). The number of states in the global system can grow very quickly with the number of processes involved (in fact exponentially). The number of states of a system made out of automata $A_1, ..., A_m$ with $n_1, ..., n_m$ states respectively, would be in the order of $n_1 \times ... \times n_m$ (and thus exponential in m). This can be a very real problem if we want to represent explicitly all states of the product system (for example in memory). This is called the *state explosion problem* and it is one of the most common obstacles for model checking large systems [5].

Let's work on an example:

Let's suppose we have a very simple model of a mechanical controller for a pair of

[5] there are methods to alleviate the problem—notably symbolic model checking methods—but it remains an important problem nonetheless

traffic lights with three states: green, orange and red. Our model just loops through the colors in that order. But what we wish is to model the global system made of two separate controllers for the two directions at the intersection of two roads. For the sake of simplicity we assume each pair of traffic lights in a direction is wired to its single controller so we do not need to explicitly synchronize them. Note also that we do not specify the time each controller stays in each state[6].

So far, nothing prevents any of the nine possible combinations (for example (green, green) or (green, red) or (red, orange) or...) to happen. These form the nine possible states of the global system.

What about transitions? In our system, we need to add "status quo" transitions for each state in each process, so that a process can truly be independent of all the others. In that global system, with the "status quo transitions" there would be 36 transitions because there are 4 outgoing transitions for each of the 9 states :

- ("do nothing", "do nothing")
- ("do nothing", "go to the next state")
- ("go to the next state", "do nothing")
- ("go to the next state", "go to the next state")

If we remove the ("do nothing", "do nothing") transition which is useless (unless we build an even bigger model by putting together our multi-process system with other systems) that still leaves us with 27 transitions (9*3).

The product of several processes can be seen as an interleaving of process "steps". One process is given the possibility to advance one step (execute a transition) then another, etc. The problem is that in our model described so far, the way the processes are interleaved is totally unspecified. One process can be given two "steps" while another one is only given one. Or even worse one process could run indefinitely while the others never run.

We need to verify that each individual process in the global system has the opportunity to run from time to time. This is called a *fairness assumption* and we will further expand the subject in section 1.2.3.

1.1.5 Synchronization

Leaving the processes totally independent from one another is not what we want in most cases (ie, it is not how it happens in the real world : traffic lights do not turn green simultaneously in both directions), so we need to synchronize them one way or another. The simplest way to do so, and this way is sometimes used in practice, is to force all processes of the global system to execute one transition at the same time. So one transition of the global system corresponds in fact to a simultaneous transition in all its processes. We call this kind of system a "synchronous system".

[6] in order to model that, the specialized "timed automata" are much more appropriate

But even though that method is indeed used in practice, many systems are neither totally independent (asynchronous) nor totally synchronous: they are mostly asynchronous but their processes still interact with each other and mutually synchronize at some points of their execution. A global system made of the product of several synchronized processes is sometimes called a "synchronized product".

Synchronization can be seen as restricting the reachable states in the global system: in our traffic lights example we do not want all the 9 above-mentioned states to be reachable (namely we do not want the green-green state to be reachable). So we cannot leave the two controllers of the opposite directions (North-South and West-East in our example) unsynchronized (which could be troublesome). One way of doing this is to prevent each of the two controllers to execute the transition from red to green unless the controller for the opposite direction executes the transition from orange to red.

We have already said that synchronization may be a restriction to the reachable states in the global system, but determining whether a state is reachable in the synchronized product or not is not always trivial. And this is an important problem in model checking because many model checking problems can be expressed in terms of reachability.

There are several ways to achieve synchronization but we will only discuss here the two most common methods: message passing and shared variables.

Message passing

In the context of model checking, one way to model synchronization is to declare a synchronization *channel* and send messages over that channel. A transition which sends a message m on a channel c will be labeled $c!m$, while a transition that waits for a message on that same channel will be labeled $c?v$, where v is the name of the variable that will store the message. A transition which sends a message on a channel will only be executed if there is another transition that is waiting for a message on that channel and is ready to be executed. One should note that channels can also be used for the sole purpose of synchronizing processes without transmitting any message (i.e. data).

In our example of the traffic light controllers, we name the channel "changedirection" (see figure 1.5). In both directions the transition from green to orange can be freely executed (it is unsynchronized). But before each process can execute the transition from red to green it makes sure the other process is ready to go from orange to red. This is done by waiting until the other process sends a notification that it is ready to execute the transition. Notice that the two automata are exactly the same except that they start in a different state.

Shared variables

Another way to synchronize two distinct processes is to let them access shared variables. Such a shared variable can, for example, represent whose turn it is to do

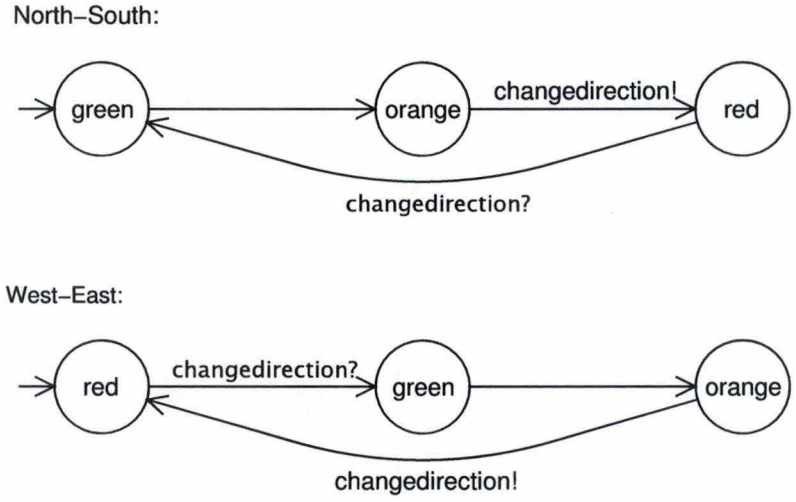

Figure 1.5: A model of traffic lights controllers using a synchronization channel

something the different processes should not do at the same time (in our example: to be in the green state).

Our example of the previous section can be transformed to use a shared variable (see figure 1.6). We use the variable *ns* to represent which direction's turn it is to be green: when *ns* equals one it is the North-South direction's turn (and obviously when it is equal to zero, it is the West-East direction's turn).

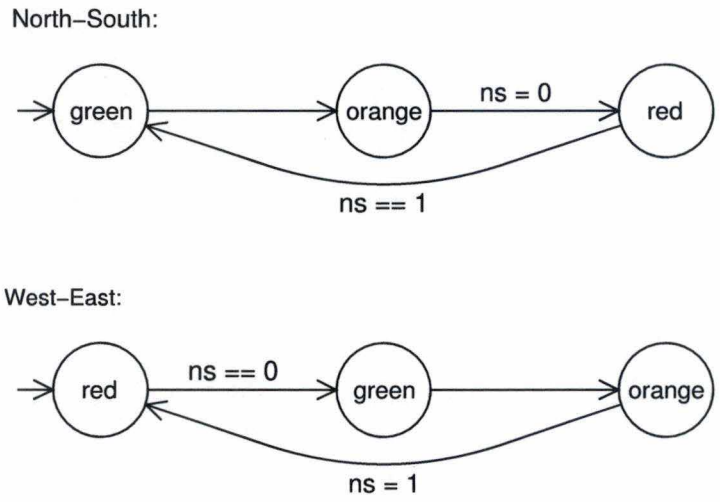

Figure 1.6: A model of traffic lights controllers using a shared variable

As we want our system to start with the traffic lights in the North-South direction being green, we initialize the *ns* variable to one. Notice that here, the two automata diverge slightly more than with the 'message passing' technique. The system's resulting behavior is also slightly different : the "synchronized transitions" occur here one after the other, whereas with 'message passing' they occur simultaneously.

1.1.6 Büchi Automata

In model checking we often need automata on infinite input. One reason for that is to allow the checking of systems that should never stop. Examples of such systems include operating systems, servers (HTTP/DNS/...), controllers (for nuclear plants/satellites/...), etc. The question is: how to define a correct behavior for those systems? Büchi automata take care of this problem by extending the notion of 'acceptable behavior' that we defined in section 1.1.2. In a Büchi Automaton, an (infinite) behavior (or run) is accepted if it passes an infinite number of times through (at least one of the states of) the accepting set. This condition is usually called the "Büchi acceptance condition".

We can define it more formally as in [8] as follow: for a determined accepting states set $F \subseteq S$, we must have $inf(\rho) \cap F \neq \emptyset$ where $inf(\rho)$ is the set of states that appear infinitely often in ρ (the set of repeating states intersects F)

Generalized Büchi Automata

Later in this work we will rather use the *Generalized Büchi Acceptance condition*, which is simply the Büchi Acceptance condition extended to multiple sets (of accepting sets) : an accepted run is, in this context, one which passes infinitely often through all of the sets. This can be expressed as : for a determined set of accepting states sets $\mathcal{F} = \{F_1, ..., F_m\}$ ($\mathcal{F} \subseteq 2^S$), we must have that for each F_i, $inf(\rho) \cap F_i \neq \emptyset$.

1.2 Properties

Nowadays it is still extremely hard (read: almost impossible) to prove that a large and complex program is completely and absolutely correct. The reason for this is that it is very hard and time-consuming to specify it completely. What we *can* do is verify only the critical parts and/or verify that the program satisfies some simple, yet very important properties. Those simple properties are often specific to the program but they can usually be categorized into families of properties, on which we will expand later on.

But in addition to the question of which properties we need to verify, there is the question of how to express them. Like for the model, the properties of a system to be verified can be expressed in different ways. It is usual though to express them using temporal logics. There are many of them but we will only present the one which is used in Divine (and is probably the most common): the *Linear Time Logic* (LTL). We will also very briefly present the *Computation Tree Logic* only for the insight it can give concerning the limitations of LTL.

1.2.1 Linear Time Logic

We will present here a common temporal logic: the Linear Time Logic (LTL) which is also called Linear Temporal Logic or Propositional Linear Temporal Logic (PLTL).

The linear time logic is used to describe properties of a particular run (see section 1.1.2) of an automaton.

LTL formulas are built by combining atomic propositions using *logical operators* and *temporal operators* (which we will detail later). Such atomic propositions are often expressed in terms of values of variables and must be evaluable to a boolean value (*true* or $false$). In an automaton using a variable x, we could for example define an atomic proposition P like this: $P \equiv (x > 1)$ which would obviously evaluate to *true* when x is strictly greater than one and to $false$ otherwise.

LTL formulas are generally evaluated over paths (the sequence of transitions taken in a particular run) and a position on that path. If an LTL formula ϕ is satisfied at a position i of a run σ of an automaton $\mathcal{A}$, we write: $\mathcal{A}, \sigma, i \models \phi$. We say that an LTL formula is satisfied on a (whole) run if, and only if, it is satisfied for position 0 on that run. But in fact what we really want to know is whether the formula is satisfied for *every* possible run of an automaton. If that is the case, we simply say that the automaton $\mathcal{A}$ satisfies the formula ϕ (and this is noted $\mathcal{A} \models \phi$).

Logical operators

The logical operators used in LTL formulas are the usual ones: $\neg$, $\vee$, $\wedge$, $\Rightarrow$ and $\Leftrightarrow$. Along with these operators LTL formulas can also make use of the boolean constants *true* and $false$.

Temporal operators

Unary operators:

- *Next*: $X\,\phi$ means that ϕ holds at the next state (the syntax $\bigcirc\,\phi$ is also sometimes used).
- *Always* (**G**lobally): $G\,\phi$ means that ϕ holds in all subsequent states of the path (the syntax $\Box\,\phi$ is also used).
- *Eventually* (in the **F**uture): $F\,\phi$ means that ϕ eventually holds somewhere on the subsequent path (also represented by $\Diamond\,\phi$).

Note that the F and G operators are dual of each others ($G\phi = \neg F \neg \phi$).

Binary operators:

- *Until*: $\phi\,U\,\psi$ means that ϕ holds until at some position ψ holds. This implies that ψ will be verified in the future.
- *Weak until*: $\phi\,W\,\psi$ means that ϕ holds until ψ holds. The difference with the *Until* operator is that there is no guarantee that ψ will ever be verified.
- *Release*: $\phi R \psi$ means ψ holds in all subsequent states unless there is a previous state in the path where ϕ holds (the syntax $\phi\,\tilde{U}\,\psi$ is also used). The name of

the operator comes from the fact that the obligation for ψ to hold is *released* by ϕ being true in a previous state. It is, in fact, equivalent to the *Weak until* operator except that the operators are inverted.

The careful reader might have noticed that the Release and Weak until operators are not the only ones to "overlap": many of these temporal operators can be defined in terms of each other. In fact, it is possible to define them all using only two of them. For example, you could use the X and U operators to define the other operators by using the following relations:

$$\begin{aligned} F\phi &= true\, U\, \phi \\ G\phi &= \neg F \neg \phi \\ \phi\, W\, \psi &= \phi\, U\, \psi \vee G\phi \\ \phi\, R\, \psi &= \psi\, U\, \phi \vee G\psi \end{aligned}$$

Formal syntax

LTL formulas can be defined using the following grammar:

$$\psi ::= p \mid \neg\phi_1 \mid \phi_1 \wedge \phi_2 \mid X\phi_1 \mid \phi_1\, U\, \phi_2$$

where $p \in P$ (the set of atomic propositions) and ϕ_1, ϕ_2 are LTL formulas.

Example

On figure 1.7, we present a totally artificial automaton (it does not represent anything in particular) to demonstrate the relation between atomic propositions and LTL formulas. Note that we labeled the automaton with the atomic propositions holding for each state and we will use these labels to designate the states.

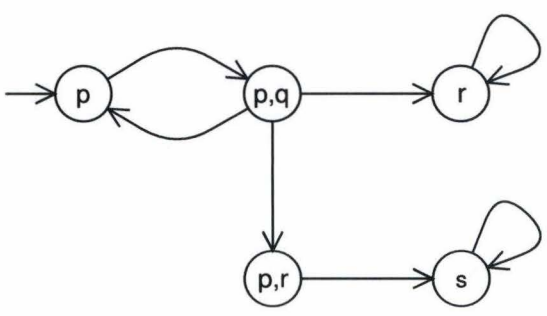

Figure 1.7: Atomic propositions on a—purely artificial—automaton

In this example, the following LTL formulas are true:

$$G(p\, W\, r) \tag{1.1}$$

It means that in all states, p will hold while r is not true.

$$G((p \wedge r) \Rightarrow X\, s) \tag{1.2}$$

It means that in all states, *if* we have p and r then at the next state s will hold.

... while these are not:

$$G(p\, U\, r) \tag{1.3}$$

It affirms that "for all states p will hold until r is true (which will happen someday)" but this is not true because the "r" state is never reached if the automaton loops between the "p" and "p,q" states.

$$F\, s \tag{1.4}$$

It affirms that "(starting in the initial state) the atomic proposition s will eventually hold" but this is not true because there are two paths (runs) where the "s" state is never reached: either if the automaton loops between "p" and "p,q" or if it loops in "r".

1.2.2 Computation Tree Logic

The main limitation of LTL is that it expresses properties of a particular run of an automaton (ie system) and not of all the possible runs of that system.

The Computation Tree Logic (CTL), also called Branching-time temporal logic, is another temporal logic commonly used in model checking but which overcomes this limitation. Instead of being evaluated over one particular path, CTL formulas are evaluated over the whole tree of paths (all the possible executions of a nondeterministic automaton can easily be arranged into a tree).

Consequently, it must have two more operators than LTL: the *path quantifiers*. These path quantifiers specify whether the (part of) formula which is under their scope must be true for *all* possible paths (operator 'A') or for (at least) *one* path (operator 'E'). Note that, as for the F and G operators, these two operators are dual of each others ($A\phi = \neg E \neg \phi$).

There is an important restriction in CTL though: temporal operators must always be immediately preceded by a path quantifier (the possible combinations are thus AF, AG, EF, EG, AX, ...). This limits the expressivity of the logic but also greatly simplify model checking with CTL formulas.

Note that even with the added path quantifiers, CTL is often considered more "limited" than LTL. Both logic have their advantages and weaknesses: LTL being considered more "powerful" and CTL being "easy" to check (the algorithm for model checking using CTL formulas is quite simple and can be done very efficiently compared to model checking with LTL).

CTL* was introduced later to combine LTL and CTL: it is a superset of both. It can either be seen as LTL with path quantifiers or as CTL without the restriction mentioned above. Even though it can express more situations than CTL and LTL, it is (currently) less popular than them (probably because of its technical complexity).

1.2.3 Families of properties

We will introduce here the usual classification for verification properties. In addition to illustrating the operators mentioned above, knowing which family of properties a particular property belongs to can be useful for several reasons:

- Not all properties of a system specification have the same importance: the cost of a property which proves not to be verified although it should have been, can vary greatly. Not only that, but the difficulty to verify one type of property or another varies too. For example safety and reachability properties are often critical to a system but they are the easiest to check.

- Another reason is that some types of properties cannot be expressed using certain kinds of logics. Furthermore different verification techniques exist for different type of properties (some techniques only work for—or are optimized for—a particular type of property).

Reachability

Reachability is a family of properties which says that a particular state (or group of states) is reachable (see section 1.1.2). In other words, it means that the system has to enter that (group of) state(s) in at least one of its possible executions (it does *not* mean that it will be reached in all its executions). Sometimes the negation of a reachability property is really what is expected from a system, but in that case it belongs to the *safety* family of properties which is detailed later...

In CTL a reachability property is usually expressed using the operators E and F like in $EF\phi$ which means that, starting from the initial state, there exists at least one path where the proposition ϕ holds somewhere on it, or in $AG(EF\phi)$ which means that in all the states of all the paths starting at the initial state, there is always at least one path *starting from that state* where the proposition ϕ holds somewhere on it.

Since the E and A path quantifiers do not exist in LTL, a reachability property cannot be expressed using LTL !!! But what LTL *can* express is the negation of a reachability property (ie a safety property). Indeed $\neg EF\phi = A\neg F\phi = AG\neg\phi$, and this can be expressed in LTL since an A quantifier is not a problem (it is as if it is implicit in all LTL formulas).

There are two types of techniques to answer a reachability question. The first one is to compute the set of all the states reachable from the initial state, and then to check whether the target state is in that set or not[7]. This is done by exploring all the different possible executions of the system (this can be visualized as taking all the different paths at the same time in parallel) and remembering which states

[7]or, in the case of the target is a group of states, whether the set of reachable states intersects the set of target states

we have traversed [8]. This technique has the great advantage that, once the set is computed, it can answer any reachability question by simply checking whether the target state is in the set or not. The main drawback of this method is that, since it explicitly constructs all the (reachable) states of the (product) system, it falls right into the *state explosion problem* seen in section 1.1.4.

The second technique is also to explore all the different possible executions of the system, but this time without remembering all the states we have been through. If the state is indeed reachable, the algorithm stops as soon as the target (group of) state(s) is reached. Since this technique does not construct the set of reachable states, it can only answer one reachability question at a time, but has the advantage of mostly avoiding the *state explosion problem.* Special care must be taken in this version to prevent looping: since this algorithm does not remember every state it has already been to, it could potentially reenter a state it has already visited without recognizing it[9].

The two techniques we have just presented share a common point: they both start at the initial state and "expand" from there (and this is usually called *forward chaining*). But one can also search the other way round (this is called *backward chaining*). In the forward chaining methods, one starts with the initial state of a system, "goes" to the next states (all the states directly reachable from there) and repeats the process until (depending on the technique) either all the states have been traversed or the target state has been reached. In the backward chaining methods, one starts at the target state and then "goes" to all the previous states (all the states that lead to the current state) and repeats the process until the initial state of the system is reached or no previous state can be found in any of the "current" states (this would mean the target state is not reachable).

Backward chaining is not as simple as forward chaining since it is not always trivial to find the "previous" states of a particular state. Furthermore, one needs to know in advance which states one wants to test for reachability. The advantage of the method lies in the fact that it offers an alternative to the forward method in case of a state explosion problem. It is also subject to that problem but, depending on the automaton and the target state, the explosion problem can be much worse with a forward chaining than with a backward chaining (or the opposite). There is no way (that we know of) to determine in advance, for a particular automaton and target state, which method will "explode" faster, so one could try one method, then if the state explosion problem prevents it to find an answer, try the other method.

Safety

A safety property is usually meant to verify that a problem never occurs (hence the name). Such a property must then always be true. Examples of safety properties could be: the temperature of the reactor of a nuclear plant is always below the

[8] In fact, when doing this technique, one does not usually compute merely the set of the reachable states but rather a graph of reachable states (ie also remembers which state leads to which state). This graph is called a *reachability graph.*

[9] but there are some techniques to prevent that by detecting such loops

critical level, or, in our traffic lights example, the lights of both directions are not green at the same time.

As already said earlier, it is the negation of a reachability property, so it can be expressed with formulas such as $AG(\neg problem)$ in CTL, and $G(\neg problem)$ in LTL.

Liveness

A liveness property is one that *must* be verified sooner or later by the system. A few examples:

- the request must eventually be satisfied,
- the light will eventually turn green,
- the program will eventually terminate.

Note that this is different from a reachability property. Indeed, we do not merely want to specify that it is possible to reach a certain state, but rather we want to specify that it will occur for sure. The difference between a reachability property and a liveness property is the same than between the W and U properties.

The temporal operator that comes to mind is the F ("Eventually") operator. So it is usually expressed in LTL with something as simple as: $F\phi$ (like in our proposition 1.4) or something slightly more complicated like: $G(requestreceived \Rightarrow Frequestsatisfied)$ which means that in all states, if the proposition "requestreceived" holds, then somewhere in a later state of the same path the proposition "requestsatisfied" holds. In CTL, the A path quantifier is usually used in addition to the F operator for liveness properties, and as such the last LTL formula would be expressed as $AG(requestreceived \Rightarrow AFrequestsatisfied)$

Fairness

Fairness is a family of properties meant to express that a particular event will happen an infinite number of times. It has only sense to define such a property on systems which have runs (behaviors) of infinite length, and therefore should not terminate.

As we already hinted in section 1.1.4, fairness properties are often used to verify that every part of a (non-terminating) system is given often enough the opportunity to execute. A fairness property can thus be seen as a repeated liveness if we repeat infinitely often a property like "that part must be executed sooner or later".

In the previous paragraph, we have used the rather ambiguous term "that part". Therefore the question is : *which* states of a system? This must be either explicitly specified into the property or can be expressed directly in the model, as when modeling systems with (generalized) Büchi automata (see section 1.1.6). In that case, the "parts" that need to repeat infinitely often are all the accepting sets of all the processes of the system.

In LTL a fairness property can be expressed as $GF\phi$ or $FG\phi$ which mean respectively that "in all states there is a guarantee that ϕ will hold at some time in the future" and "ϕ will hold forever starting from a point in the future". Expressing a

fairness property in CTL can be problematic in some cases because of its restriction of having a path quantifier directly in front of every temporal operator. This problem is usually overcome by slightly extending the logic but we will not discuss that in this paper.

Sometimes we want to consider fairness as an hypothesis rather than as one of the properties to be verified, for example if one is asking himself : "for a *fair* execution of that system, is that property verified?". This is called fair model checking as in "fair LTL model checking". It is interesting to note, as stated for example in [4], that "fair LTL model checking can be transformed into normal LTL model checking by putting the fairness requirements into the formula". In that case the new LTL formula to verify is something like :

$$\psi = (\mathrm{GF}F_1 \wedge ... \wedge \mathrm{GF}F_n) \Rightarrow \phi$$

where ϕ is the "old" LTL formula (the formula we want to check under a fairness hypothesis), and F_i is an atomic proposition stating that "the state the system is currently in" is part of the ith part of the system that needs to be visited infinitely often. This new formula thus states that if all the designated parts of the system are visited infinitely often, then ϕ holds.

If we are using Büchi automata to model our system, the above LTL formula would become

$$\psi = (\mathrm{GF}F_1^1 \wedge ... \wedge \mathrm{GF}F_n^m) \Rightarrow \phi$$

where F_j^i is the jth accepting set for the ith process. In that case, ψ (the "new" formula) states that if all accepting sets of all processes are visited infinitely often, then ϕ holds.

Deadlock freeness

One property which is generic and that most (if not all?) programs want is the deadlock freeness property. We will use the following definition of a deadlock : a system is in a deadlock if it is in a (non-accepting) state where no transition is executable. The usual definition for a deadlock [10] is that several competing actions are awaiting each other's execution. Since they are all waiting for each other, none of them can execute.

This usually happens when several processes (1) share a global mutually-exclusive resource, (2) need several "parts" of it to execute a particular action (actually blocking the process while these are not available), and (3) acquire these "parts" progressively (not all of them at the same time).

The most classical example of this is probably the dining philosophers: 5 philosophers are at a round table on which 5 forks are placed. Each of them need two forks to eat. So they take one fork then another one and release both of them when they have finished eating. If they all take their first fork at the same time, none of them can eat since they each have exactly one fork in hand (and all miss a second fork).

[10]The definition we just gave is broader than this "usual" one but it is this "broader" definition most programs should be free of

They will wait like that forever since each one of them will wait for some other to finish eating (to grab one of his forks) but this cannot happen. They are thus in a state of deadlock.

What about verifying the deadlock freeness of a system? In CTL, the deadlock freeness can be expressed in all generality (i.e. this is valid for any system) by a formula like $AGEXtrue$ which means that in all the positions of all the possible executions of the system, there is at least one executable transition.

In LTL, such a general expression (valid for any system) is not possible. But the deadlock freeness of a *particular* system can be expressed. Indeed, in a particular system, the conditions for a deadlock to occur or even better, the states in which it occurs, are usually known. A deadlock freeness in such a system can thus be expressed by a safety property (stating that the conditions never occur or that the states are never reached).

In the dining philosophers example, let us suppose we have the atomic propositions $P1$ through $P5$, each of them meaning that the philosopher with that number has exactly one fork in hand. In that case, a deadlock freeness property can be expressed in LTL by $G\neg(P1 \wedge P2 \wedge P3 \wedge P4 \wedge P5)$.

1.3 Verification

In the previous sections we have explained how to model a system, which kinds of properties one could verify and how to formulate these properties. What remains to be seen is how to verify that the model satisfies the properties. One possibility to do that is to use an *observer automaton.*

1.3.1 Observer automata

An *observer automaton* (also called property automaton) is a special automaton which can be affixed to a system to *observe* it (hence the name). The originality is that it does not represent another process of the system and as such it does not expand the functionality of the system. One might wonder what an observer automaton observes. The answer is that it observes whether the system it is attached to satisfies a particular property or not.

In order to perform its task properly, an observer automaton needs to be "omniscient" about the system: it needs to know about every transition in every process of the system. For that to be possible it must be synchronized with every process of the system: when any process executes a transition, the observer executes one too (possibly a transition with no effect whatsoever and leading to the same state). Note that this synchronization between the observer and the other processes does *not* mean that the processes have to be synchronized with each other.

Another originality about observer automata is that their *transitions* are not labeled with standard expressions, but rather with the atomic propositions that hold in the different *states* of the system. They can thus observe the sequence of these atomic propositions during an execution of the system being checked.

A property is satisfied for a particular run of a system if, for that run, the observer automaton remains indefinitely in one of its accepting states without blocking (meaning that it can execute a transition for every transition executed by the processes of the system).

An interesting fact is that one can automatically construct an observer automaton corresponding to an LTL formula. We shall not present the details of such a construction in this paper, but will rather show an example: figure 1.8 depicts an automaton accepting the following LTL formula:

$$F((p \wedge r) \wedge X \neg s)$$

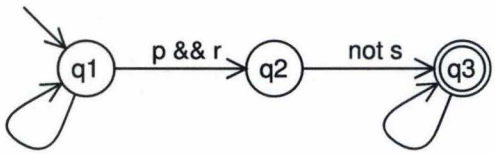

Figure 1.8: An observer automaton accepting an LTL formula

As an example of what a "real" observer (that is an observer of a particular system) is, and to illustrate the fact that atomic propositions are in fact evaluations of variables, let us present the same automaton with the atomic propositions p, r and s defined as follows :

$$p : x == 0$$
$$r : y == 1$$
$$s : y > 1$$

The resulting automaton is shown in figure 1.9.

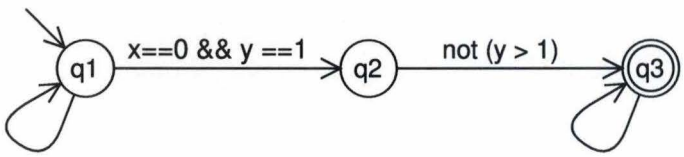

Figure 1.9: The same automaton with atomic propositions expressed as variables

One should note that the fact that an observer automaton can be constructed from an LTL formula does not mean observer automata can *only* be constructed from LTL formulas (or other formalisms). An observer automaton can also be constructed directly "by hand" to express a property.

1.3.2 Algorithm

Thus far, we have seen how to check that a particular run of a system satisfies an LTL formula. What we would like to know now is whether *all* the possible runs of

the system satisfy that LTL formula. At first, this problem could seem hard to solve but it can be achieved quite simply by the following "algorithm":

- Negate the LTL formula. For example, the LTL formula that we have used earlier in section 1.2.1:

  $$G((p \wedge r) \Rightarrow X\, s)$$

  negates to:

  $$F((p \wedge r) \wedge X \neg s)$$

- Construct an automaton accepting that negated formula. The automaton for our example formula corresponds to the one on figure 1.8 (or rather figure 1.9 if we define the atomic propositions p, r and s the same way than in the previous section).

- Use that automaton as an observer automaton for the system.

- If the observer automaton "accepts" the negated formula then that means there is at least one run of the system where the original LTL formula is violated.

- If, on the contrary, the observer automaton does not accept the negated formula that means *all* the possible runs of the system satisfy the original LTL formula!

This concludes this first chapter where we attempted to outline the various facets of Model Checking, some of its limitations, the properties that are the subject of Model Checking, and the means of verifying these.

Chapter 2

Visualization

In this chapter we will first introduce the reader to some of the possible extensions of the graphical representation of concurrent systems we used in the first chapter of this work. Then we will briefly talk about the need for and the use of graphical editors in the context of model checking and the features they could (or should) have. Finally we will examine what kinds of graphical editors for finite state machines exist in the context of verification tools and also outside that context.

2.1 Visual formalisms

As we already stated, visualizing the model of a system greatly helps most people to grasp its mechanics. But the graphical representation seen in the first chapter was quite limited in several aspects. Therefore, we will attempt to go beyond those limitations by presenting in this section the *visual aspect* of three formalisms used in the domain of concurrent systems. The aim is only to give the reader an idea of what can be done. We will therefore limit ourselves to outlining the most important features of these formalisms and explaining them informally (thus avoiding rigorous semantics for them).

First, let us be aware that complex systems are often not modeled as we did in the first chapter. Indeed, if the size of the system increases, either each process grows so much that it becomes unpractical to model them, or the number of concurrent processes explodes and it then becomes almost impossible for one to have a global view of the whole system.

What usually happens is that complex systems are split into several mostly independent parts that we shall call *components*. From what we have seen up to this point, those components could be the processes we have outlined in chapter one. But when the system is large, these components are further divided into subcomponents. And that process of dividing (sub)components is repeated until each of them has a manageable size.

The problem of modeling complex, possibly communicating, systems can thus be divided into three distinct problems:

1. how to model the *hierarchy* of components of the program. That is how the

system has been divided into components and subcomponents.

2. how to model the *internal behavior* (the control flow) of those components.

3. how to model the *communication* (or coordination) between components.

The first problem is entirely new and we will try to give the reader an idea of how it can be addressed by presenting two different formalisms: *statecharts* and the *Darwin architecture language.*

The second problem has already been discussed in the first chapter. Indeed, the internal behavior of components is often depicted by some sort of extension (or variant) of Finite State Machines. Therefore we will not discuss this topic any further in this chapter except for the fact that statecharts model both the hierarchy and internal behavior of components.

The third problem was already discussed in the first chapter (in the synchronization of concurrent processes) but the visualization of the communication was non-existent. We will present here the Message Sequence Chart formalism as an example of how the communication can be explicitly visualized. Additionally the Darwin architecture language addresses the problem too.

Amongst the numerous visual formalisms that exist we will only present those three, not because they are the most widely used (this is probably not the case), but rather because most of the other (visual) formalisms we know of are similar to one of these or a combination thereof. We are convinced that the three we outline are a good representative sample.

2.1.1 Statecharts

To overcome the limitation of modeling complex systems with possibly-communicating finite state machines, statecharts were presented by David Harel[1]. He later summarized them in [3].

Statecharts can be seen as an extension of Mealy machines (standard finite state machines with their output tied to their transitions). The extension consist mainly of the introduction of three new concepts: depth, orthogonality and broadcast communication.

Depth

Depth is the concept of hierarchy itself, that we already explained above, in a "top-down" approach (starting from the program as a whole and then dividing it). The equivalent "bottom-up" approach would be to say that several states can be grouped into a component. And that components can be further grouped into larger components. This is analogous to the "folding" principle seen in the first chapter where we grouped several states together in one "virtual" state.

Having a hierarchy of components would be of little use if it was not possible to have transitions from and to components (instead of only from and to states). But

[1] in his paper "Statecharts: A Visual Formalism for Complex Systems"

these transitions are ambiguous: what does a transition from a state to a component, or from a component to a state, exactly mean?

To resolve the ambiguity of a transition from a state to a component, it suffices to specify an initial state for that component. A transition from an external state to the component will, in fact, be a transition to the initial state of the component.

The ambiguity concerning the outgoing transitions (from a component to a state) can be lifted if we only allow these transitions to be defined if all the states of the component have a transition to the target state. A transition from a component to an external state will then be a transition from the "current" state of the component to the external state.

Note that it is still possible to "bypass the component" and have transitions directly from or to internal states of the components.

Orthogonality

Orthogonality is a concept which allows several concurrent components (or states) to be regrouped into one component. These concurrent components are called *orthogonal components*. Having several orthogonal components regrouped as a single component can be seen as regrouping several concurrent processes into one "virtual" state. Note that each of these orthogonal components need an initial state, as all the other components.

Broadcast communication

Broadcast communication can be described as the fact that output events of transitions are "broadcasted" to the whole statechart, in addition to being sent to the "outside world". These output events can thus trigger other transitions inside the statechart and cause a so-called *chain reaction* (in David Harel's own words).

Example

All these concepts are (at least partially) illustrated by the example[2] depicted on figure 2.1. In this example we modeled a simple beeper (or, if one prefers, this can be seen as a model for the process of receiving text messages in mobile phones).

Our beeper consists of only two main states: `ON` and `OFF`, but the `ON` state is a "grouped state", made out of three *orthogonal components*: a Receiver component, a Screen component and a Sound component, which are separated on the figure by dashed lines. In our model, the beeper starts in the `OFF` state. When it is turned on, it goes to the `ON` state and all the "orthogonal components" that component is made of start in their respective initial state: the Receiver component in its `WAITING MESSAGE` state, the Screen and the Sound components in their respective `IDLE` state.

Then, when a message starts to be received (i.e. the `m_start` event is received) two things happen: the Sound component moves to its `BEEP` state and the Receiver component moves to its `RECEIVING MESSAGE` state. Then, when the Sound component

[2]this example is roughly inspired from an example found in [7]

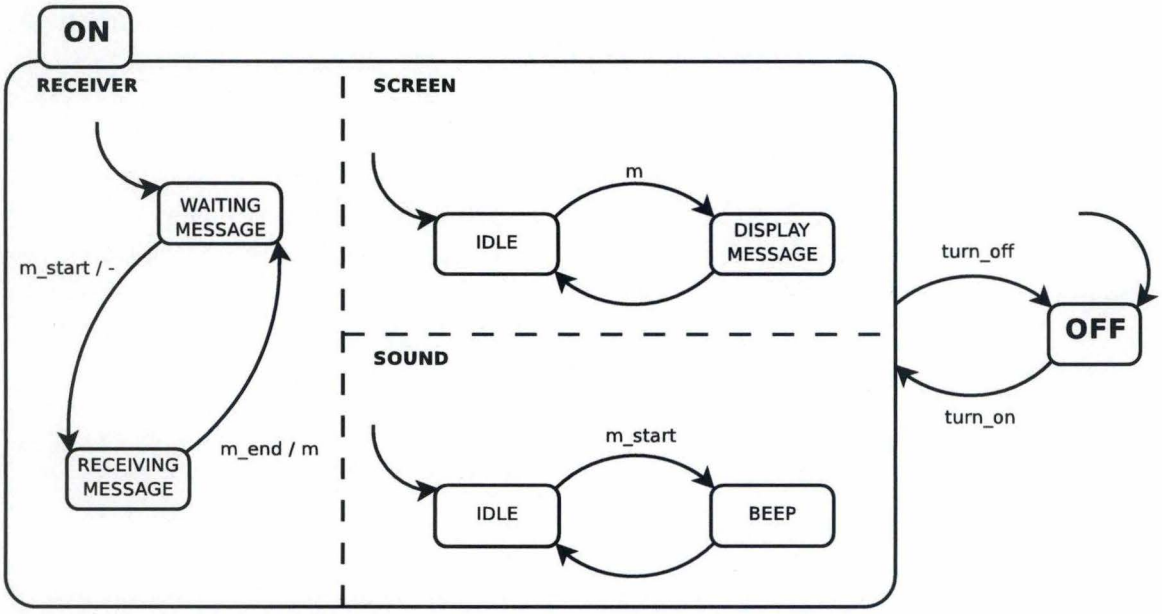

Figure 2.1: A model of a beeper using the statechart formalism

has finished "beeping", it goes back to its `IDLE` state. Likewise, when the Receiver component has finished to receive the message (which, unlike for the Sound component, depends on "the outside", i.e. it must have received an `m_end` event), it goes back to its `WAITING MESSAGE` state. But here something interesting happens: when that last transition is triggered by the `m_end` event, it broadcasts the received message (`m`), and this causes in turn the Screen process to execute the transition from its `IDLE` state to its `DISPLAY MESSAGE` state. This is a *chain reaction* of length 2 (there can be chain reactions of arbitrary length—which is one of the shortcomings often cited about statecharts).

2.1.2 Darwin

Darwin [2] (or, officially ∂arwin) is a language meant to describe component hierarchies and their interactions (which are a form of communication). It does not address the internal behavior of the components at all. This approach is sometimes called "programming-in-the-large" or "black box" approach: the internal behavior of the components used as the building blocks for the model need not to be known by the person modeling the system, as if they were in a black box.

In the visual representation of Darwin, components are represented by boxes. Darwin also introduces the concept of *service*. A service is some form of inter-component communication. A service will always have two classes of components dealing with it: the components which *require* it, and the components which *provide* it. The graphical representation of these are white dots for *required services* and dark dots for *provided services*.

Example

Figure 2.2 shows our model for a simple multimedia library. This library *provides* functionalities for playing raw audio samples, music and movies. It relies on (*requires*) an audio and video driver for the actual interaction with the hardware (the sound card and video card respectively) to ultimately play the raw audio data and display the video data. Such a library could, for example, be used as a back-end in a multimedia player.

Our multimedia library component is further divided into subcomponents which help provide its functionalities. In our example, the library does not need any subcomponent to play a simple sound: it can forward it straight to the audio driver. But playing a music file cannot be done the same way: the music first needs to be decoded (in the *Music decoder* (sub)component) before being sent to the audio driver. Playing a movie is even more complicated: the data first needs to be separated into audio and video data by a *Music/video splitter* component. But this subcomponent does not know how to decode those audio and video data, so it calls upon the *Music decoder* and *Video decoder* components. Like the *Music decoder* component, the *Video decoder* component does not know itself how to display the now-decoded video data so it calls upon the video driver.

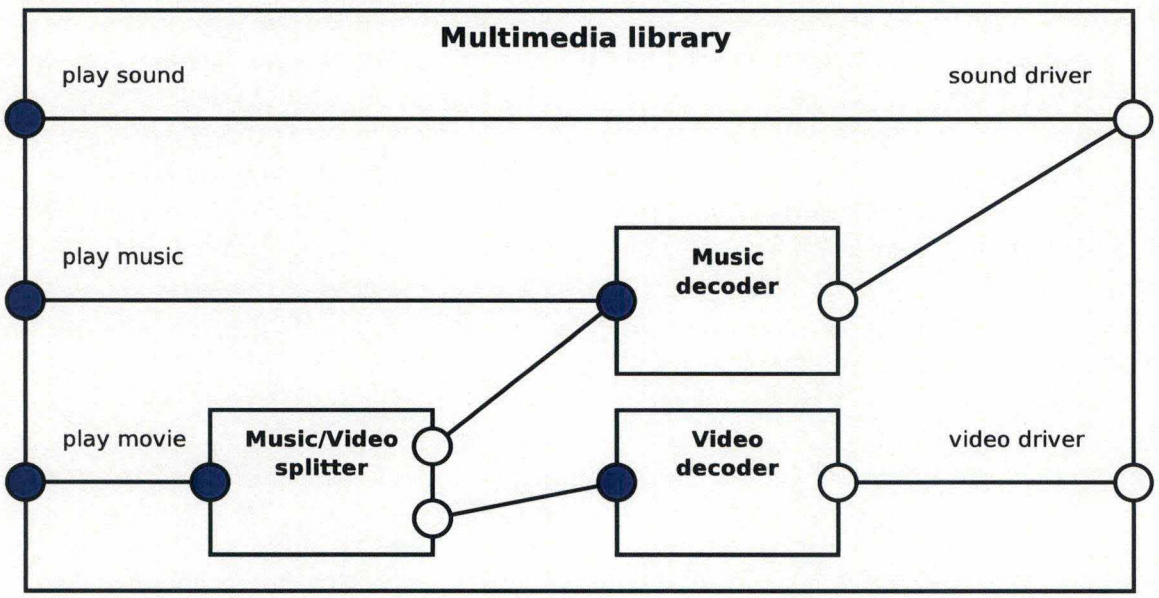

Figure 2.2: A model of a multimedia library

Black box modeling?

One interesting feature of such a visual formalism is that one can easily take the "black box" approach one step further: use components to build larger components without taking their internal structure into account. The difference with what was explained above is that one can even omit the subcomponents of a component (not

only the internal behavior of the "atomic components", as we explained). An example of this is shown on figure 2.3. This can of course also be achieved with other formalisms but special care must be taken if the formalism allows to model transitions (or communications) directly from or to internal states (or subcomponents) of a component, as is the case for the statecharts formalism presented above.

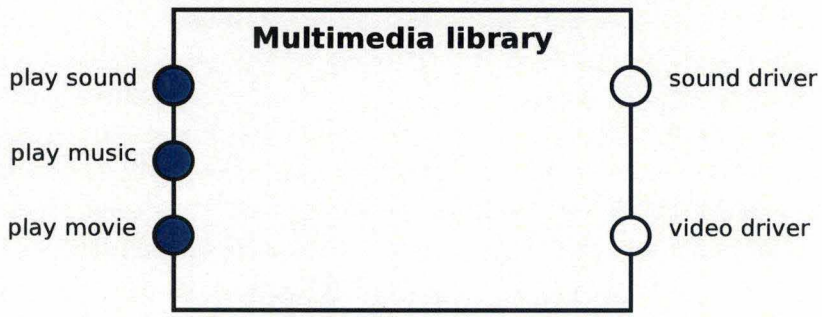

Figure 2.3: A model of a multimedia library, using a "blackbox" approach

2.1.3 Message Sequence Charts

Message Sequence Charts (MSC) [5] is a data-flow based formalism. This means it is concerned with the exchange of messages over time. These messages are exchanged between concurrent processes. The formalism is mainly concerned about two aspects: the sequence of messages exchanged, and which processes are involved in each exchange.

The graphical representations of Message Sequence Charts vary widely, but mostly involve drawing a vertical line for each process : going down these lines means advancing in time. Then, each message is represented by an arrow from the line corresponding to the process sending the message, to the line corresponding to the process receiving the message.

Message sequence charts are particularly suited to visualize the sequence of messages exchanged in a particular run of a system made out of concurrent processes (like what was described in the first chapter). A direct application of this is the visualization of traces of simulations of systems that is offered by some tools like UPPAAL[3].

Example

In the figure 2.4 we show a message sequence chart for our traffic light example from chapter one. Note that we have taken some liberties compared to the "official" formalism[4] : we represented the states on the sides and only meant them as time reference points.

[3] `http://www.uppaal.com/`

[4] it seems to be a common practice "to take liberties" about this formalism. Our guess is that this might be due to the fact that the semantics for some of its elements were not standardized for some time, and several different uses were developed for the same elements.

One should note that Message Sequence Charts can be of arbitrary length since they can represent a "run of a system" and, as we have seen in the first chapter, such a run can be of infinite length (even for a system with a finite number of states).

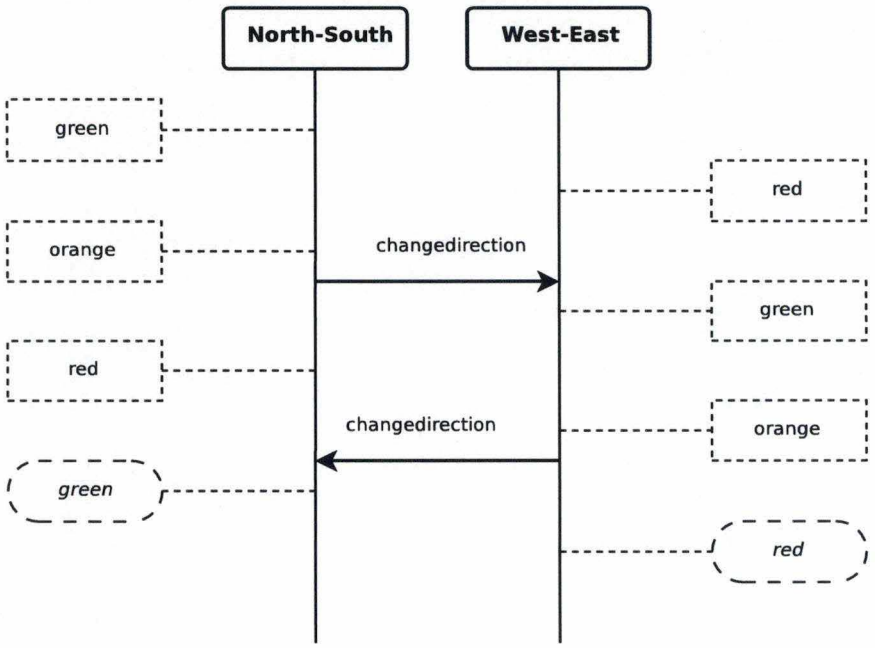

Figure 2.4: A Message Sequence Chart for the synchronization of traffic lights

2.2 Graphical editor

In this section, we will touch upon two main topics : what can a user wish for from a graphical editor for concurrent systems, and what is already available in that domain. But, in order to determine the features needed by such a graphical editor we will, at first, be more general and go through some of the possible uses of verification tools.

2.2.1 Usages of verification tools

We shall distinguish two types of uses for verification tools: we shall call them, for the purpose of this paper, "post-implementation verification" and "pre-implementation verification".

Post-implementation verification

By the term *post-implementation verification* we mean "to verify existing systems". The best way to do so (or, at least, the easiest way) is probably to automate the process of describing the system in a language understandable by the verification tool. This can be done by having an automatic converter from "the source code"

of the system to our modeling language. By source code, we do not only mean the source code from a software program but also the (detailed) description of the hardware circuit that was used to engineer it.[5] Note that tools to perform such automatic conversions already exist for many programming languages (and probably for "hardware description" languages too).

When verifying an existing system, a Graphical User Interface (GUI) is therefore not vital and the same applies to a graphical editor. Nevertheless those can be very useful for several reasons:

- First, for visualizing what is being checked. Most systems of a certain complexity are first modeled then implemented. If one wants to check properties of that model, one will need to convert it to a formalism understandable by the model checker. In that case, it would be a good idea to first make sure that the converted system is indeed what is expected before doing the actual model checking. And such a validation step can be done much easier if done visually.

- Secondly, in a process of iterative verification and correction of a model, it is probably easier to find where the properties were not verified on the model than it is to find the corresponding location in the source code. Indeed, many verification tools provide the user with "counter-examples" for properties that are not verified. By counter-example, we mean an example run of the system where one property is not verified. In that case, and before proceeding to the verification of other properties, it is much easier to correct the "verification" model directly by editing it graphically . In this case, a user could also wish to export the resulting model to a programming language, but this exporting capability of verification tools has little to do with the graphical user interface.

Pre-implementation verification

By the term *pre-implementation verification* we mean "to verify systems that are not yet implemented nor fully designed".

If the visual formalism of the verification tool is rich enough, one could take the final design step for such system directly with the verification tool[6]. This could really benefit the design process because the design can be checked directly as soon as it is modeled, or even parts of it could be verified during the modeling. In this case too, being able to automatically convert the model to a "production format" (for example source code in a classical programming language) is a must-have feature.

A drawback to this approach is that one could prefer to use an integrated tool for all the steps of designing a system, like in the numerous UML-based tools.

So far, we have presupposed that all the systems to be verified would be extremely large and complex. While this is often true, it is far from being always the case.

[5] we believe that, nowadays, virtually no hardware system is designed without computer aid. Therefore, we believe that the assumption that there is always a description of the hardware system in one language or another is reasonable.

[6] we assume a several-step design process, which is the standard in the industry nowadays.

Some systems can be of critical importance (and thus needing verification) while being relatively small. Communication protocols often enter this category. Such "small" systems could be (reasonably) easily modeled even with a relatively poor visual formalism (as, for example, the one presented in the first chapter).

In addition to all that was already said, a graphical user interface (and especially a graphical editor) can be very useful for students and newcomers to the domain in general. Indeed it is much more appealing, especially for someone who does not master the domain, to work with graphical interfaces rather than command-line driven tools.

2.2.2 Graphical User Interface

We will summarize here what the different usages of verification tools outlined above imply in terms of features for a graphical user interface (GUI). The actions itemized below can be taken as a partial list of *usecases*.

Note that verification tools are especially suited for the classical dichotomy between a front-end (the graphical user interface) and a back-end (the actual verifier). In the following text, we will assume such a back-end is at our disposal and only focus on the GUI. But even in that case, the user interface should still give the user the possibility to use all the functionalities of the back-end.

We can categorize the needed features using the same classification as the one we used in the first chapter: modeling, specifying properties and verifying that the model satisfies the properties.

Building a model of a system

The interface should allow to do it in three ways:

- Creating a new model from scratch
- Creating a model by importing it from an external source
- Loading an existing model (already made with the editor) and modifying it. This obviously implies that a "save" functionality must be provided by the editor.

In the topic of designing a graphical editor, which is what we are after, the most important in the above list are : to be able to create a new model from scratch, and editing an existing model. This will be examined in the next chapter.

Specify the properties to be verified

The interface should allow the user to specify such properties within the editor using one of the logics supported by the "back-end" (for example, the Linear Time Logic seen in the first chapter). In addition, it could allow the user to describe properties as an observer automaton.

Validate the model

Such an editor should be able to interact in several ways with the back-end to validate the model. The most obvious way is to "launch" the model checker to verify that the model satisfies the specified properties. But another interesting way to validate a model is to simulate it: "run" it step-by-step (or rather transition-by-transition) and "see what happens".

Export

As already mentioned, another desirable feature is to let the user export a model to a programming language or to a "format suitable for engineering hardware", even though such a feature is, strictly speaking, out of the scope of *verification* tools.

2.2.3 State of the art

We will present the reader with an overview of what exists in terms of graphical editors for concurrent systems. First let us precise the scope of our overview: we will only present here editors which use a formalism close to Finite State Machines, and which are freely available. This second restriction probably places a strong bias on the results of our "search", but the reader may understand that we cannot talk about a particular tool without having tried it.

Note also that we compare only the graphical editors (and, to a lesser extent, the user interface and the functionalities it provides) but *not* the verification tools of which some of these editors are a part.[7] Therefore we will not talk about the formalism used to express properties, nor will we be very rigorous about the modeling formalism used by those tools (we are only interested in their visual aspect).

We pursue two distinct goals :

First, we simply want to "observe the competition". This should help to bring some perspective to the work which we present in the next chapter.

Secondly, we want to find out if there are open-source editors that could be (or could have been) extended in order to create an editor for concurrent systems. Note that this question was, unfortunately, only looked into a posteriori (i.e. after we developed our own editor). Therefore we provide here the results of our search only for the added perspective it can give.

Unless otherwise stated all the following tools are both free (our original requirement) and open-source (and thus could be extended).

We have divided the tools into three categories: Graphical Editors in Verification Tools, Graphical Editors in other contexts, and Libraries. The last two categories are here mainly in order to fulfill our second goal.

[7] for such a comparison, the reader is invited to consult the YAHODA Verification Tools Database - http://anna.fi.muni.cz/yahoda/

Graphical Editors in Verification Tools

The situation is not very bright concerning graphical editors in freely available verification tools. And this also applies to graphical user interfaces in general. To quote [1]:

> "[...] to consider only tools issued from universities does have some consequences. They all present some very typical limitations: [...], the graphical user interface is sometimes very rude, etc. In general, the authors of these tools preferred to develop an original algorithm to solve a difficult problem rather than enhance the ergonomics or portability."

Even if we did not limit ourselves explicitly to tools issued from universities, we did limit ourselves to freely available tools. Since most of the freely available tools in the field originate from universities, we fall in exactly the same situation. Some of these verification tools provide a graphical user interface but only very few of them contain a graphical editor (i.e. to let the user graphically create or modify models).

We could only find one tool with a truly appealing interface: UPPAAL[8]. It is Java-based and integrates a graphical system editor and a graphical simulator (which includes a Message Sequence Chart-based visualization for its traces). It also allows the user to define and check properties in a very user-friendly way. The graphical system editor in itself provides close to everything a user could expect from such an editor: most components (notably states) are *draggable* and have a "context menu", transitions can be added by simply clicking on the states they "link", the drawing area is *zoomable*, etc. Unfortunately, UPPAAL is not open-source...

The only other tools which have a graphical editor worth mentioning are: Autograph and PEP.

Autograph[9] is the graphical editor associated with the *Fc2tools*). It uses a visual formalism which could be informally described as "Darwin with Finite State Machines", the latter describing the internal behavior of components and being drawn inside the component boxes. Its user interface is rather unique in the fact that it is intuitive (easy to learn) yet at the same time agonizingly painful to use. We will also note that it is not open-source.

PEP[10] is an open-source collection of tools mainly based on *petri-nets*. In addition to a petri-net editor, it includes an editor for finite state machines. This one has a reasonable set of features and a quite usable interface but it is not extremely appealing visually.

In the area of verification tools, what seems to be a bit more common than graphical editors is *static visualization* (with no graphical edition possible) like, for example, in the nice LTSA tool.[11]

A last point to note is that we could not find any graphical editor for (generalized) Büchi automata. There are mentions of such editors in some papers, but the editors

[8] http://www.uppaal.com/

[9] http://www-sop.inria.fr/meije/verification/

[10] http://parsys.informatik.uni-oldenburg.de/~pep/

[11] http://www.doc.ic.ac.uk/~jnm/book/

they are referring to seem to be either built into commercial tools (and thus not freely available) or were not even implemented.

Graphical Editors in other contexts

For other purposes than verification tools, the situation of graphical editors for finite state machines is totally different. There are numerous fully-featured freely-available (and even open-source) graphical editors for Finite State Machines (or similar formalisms).

Many of these fall into the category of (or are incorporated into) software modeling tools. The UML (Universal Modeling Language)[12] tools alone are too numerous to even list them.

There are also many stand-alone editors like, for example, Qfsm[13] for Finite State Machines or Stephen Edwards's Editor for the Statecharts formalism[14].

A third category which seem to be plentiful are the "language tools". These often offer two interesting features: the possibility to construct an automaton automatically to accept a particular regular expression, or to simulate runs of a modeled automaton to verify that an expression satisfies a particular grammar. The best example of this category is probably jrexx-Lab[15], a user-friendly and very polished graphical editor for regular expressions.

Finally, we can say that the programmer who still has not found what he likes in what was presented above, can still extend his search a step further by looking for editors for directed graphs, which are even more plentiful.

Libraries

Libraries that could facilitate the creation of a graphical editor for concurrent systems are numerous too. There are mainly two types of these: libraries for drawing connected graphs, and zooming frameworks. By zooming framework, we mean a library which provides a zooming feature for applications.

We will present hereafter only the libraries that caught our attention, and only highlight what we believe is their most relevant feature in the context of creating a graphical editor for concurrent systems:

- JGraph[16]: a seemingly quite known Java library which provides a Swing component to display and edit graphs. Note that Swing components are one of the two families of built-in graphical components in Java. One can create classes which inherit from them to extend them, which is the case of JGraph.

[12] UML can be seen as a superset for several formalisms, of which, among others, statecharts
[13] http://qfsm.sourceforge.net/
[14] http://www1.cs.columbia.edu/~sedwards/sc/
[15] http://jrexx.karneim.com/
[16] http://www.jgraph.com

- OpenJGraph[17]: another Java library to manipulate graphs (including adding, moving and removing vertexes). Despite the name it seems to be unrelated to JGraph.
- Piccolo[18]: a generic 2D graphics framework for Java and C# featuring namely "Zoomable" components.
- GINY[19]: a framework, built on top of Piccolo, featuring inter-alia several automatic layout algorithms. An automatic layout algorithm can be very useful when trying to visualize (or import into an editor) graphs expressed in a "non-visual" formalism. Such graphs would thus not contain position information for its vertexes, hence the use of a layout algorithm to place these vertexes as well as possible.
- Grappa[20]: A "static" visualization library. Like its ancestor GraphViz (probably better known as "dot"), it has a very good layout algorithm.

We will end this overview of existing libraries and conclude this chapter by mentioning the quite peculiar approach of GenGED[21]. It is not a library but still falls into the category of tools that could help to create a graphical editor for concurrent systems. Indeed GenGED stands for "Generation of Graphical Environments for Design" and, as that name suggests, it is a generator of graphical editors. It takes the description of a visual modeling language as input and generates a graphical editor for that modeling language (and a simulator for the models built with the editor). Although we are very impressed with the idea, the complexity of the program seems so high that we wonder if it really is more efficient to learn how to use the program (and build an editor with it), than to adapt an existing editor...

[17] http://openjgraph.sourceforge.net/
[18] http://www.cs.umd.edu/hcil/jazz/
[19] http://csbi.sourceforge.net/
[20] http://www.research.att.com/sw/tools/graphviz/packages/grappa.html
[21] http://tfs.cs.tu-berlin.de/~genged/

Chapter 3

Implementation

In this chapter we will present our editor for concurrent systems, Paxion. We will begin by a short overview thereof, then describe its functionalities through an example of the modeling of a system with it. Then we will describe the architecture of the program, including a short description of all its classes. Thereafter, we will outline the way we implemented the most interesting features. Note that we will not evaluate our work in this chapter as one might expect. The possible improvements of our editor, whether to correct errors we may have made or to propose possible extensions to it will be the topic of the fourth chapter.

3.1 Overview

Paxion is an editor for Networks of Extended Finite State Machines (EFSM) with an optional observer process. These Extended Finite State Machines could be defined as generalized Büchi automata extended with all the concepts we have seen in the first chapter, namely: variables, guards, effects and synchronization over channels. Additionally we support both global and local variables.

Paxion was designed to work as a (sort of) front-end to DiVinE[1] but it can also be used as a stand-alone tool.

We will not venture to qualify our editor as a complete and mature product. Indeed, it has several limitations.

Out of the features listed in section 2.2.2, we do not provide the possibility to create a model by importing it from an external source. The only possibility to specify a "verification property" through the editor is by explicitly building an observer automaton. And the only interaction with the "back-end" in order to validate a model is through exporting the model in the .dve file format (the format used by DiVinE). We will expand on some of these limitations in the fourth chapter hereunder. Even with those limitations and although it obviously does not qualify as a "commercial-quality" product, we do consider it as more than a mere prototype. As already said, we will discuss more at length the topic of "missing features" in the next chapter.

[1] a Distributed Verification Environment - http://anna.fi.muni.cz/divine/

The above premices having been set, let us now outline some general facts about this editor:

Paxion is written in Java. This has two main consequences: on the one hand, it is portable to all the platforms that Java supports (officially the Windows, Linux and Solaris Operating Systems); on the other hand, Paxion needs a *Java virtual machine* to be able to run.[2]

Our editor makes use of the *Apache Xerces2 parser*[3] and the *Java Look and Feel Graphics Repository*[4]. Both of these products are free to use and to redistribute for mostly any purpose providing one respects certain elementary conditions. Note that both of these libraries are distributed within the packaged releases of Paxion.

The *Xerces2 parser* is an XML parser which can be used through the SAX interface defined in Java.[5], while the *Java Look and Feel Graphics Repository* is a collection of icons that we use in our menus and toolbar.

Concerning the interface, our aim was to create one which would be both user-friendly for the newcomer and efficient for the regular user. As we will see later, there are specific points of the interface where these criteria were not met.

As for the graphical representation of the automata themselves, we used the usual "circle-based" representation. We have deviated slightly from the usual representation concerning initial states and accepting sets : we will show this in the example of modeling hereafter, and also discuss it in the next chapter since we consider it could be improved.

The architecture of Paxion was designed with two complementary ideas in mind: maintainability and expandability. As Java is an object-oriented language, we tried to use it to its full potential by adopting a strongly object-oriented programming style. Of course, even with those objectives in mind, the current version of our editor is not perfect in that matter, and thus still contains room for improvements.

The code of Paxion is documented following the standards of the Javadoc tool [6]. This tool allows the automatic extraction of comments from the Java source code to generate HTML documentation, provided that those comments follow a particular syntax[6]. Note that the code is not (yet) fully documented. All the classes and most of the more "tricky" parts of the program are documented, but all the methods from all the classes were not systematically documented.

---

[2]and this will be the case until compilers from Java to native platform binary code—which are in development—can compile Paxion.

[3]http://xml.apache.org/xerces2-j

[4]http://java.sun.com/developer/techDocs/hi/repository/

[5]this is a standardized interface meant to let a programmer use many different parsers using the same API

[6]But, for generating the documentation in our "release packages", we preferred to use Doxygen (http://www.doxygen.org/), a documentation system which recognizes the javadoc syntax, can generate documentation in many formats, and can also generate very nice inheritance and collaboration diagrams. To do so it uses Graphviz (http://www.graphviz.org). All the graphs in the B appendix were generated with Doxygen (some of them being tweaked afterwards by using Graphviz directly).

3.1.1 Feature summary

Paxion main features are:

- A tabbed panel to represent processes (one tab for each process).
- A *zoomable* "drawing" area. The drawing area is meant to "draw" automata.
- Side panels to specify the properties of the system being modeled and each of its processes. **Note** that we use here the term "properties" in its general meaning and *not* in the meaning we have used so far it this paper (i.e. properties to be verified by the model checker). In the rest of this work we will name these panels "system properties panel" and "process properties panel".
- Easy creation of new transitions (by clicking on the states they link). Additionally these transitions can have intermediate points (often called "nails" in similar programs) and in that case are shown as curved.
- *Draggable* components (states, labels and intermediate points).
- Automatic update of components when an object they are linked to is moved, even if indirectly (as, for example, when a state is moved, all the transitions from and to that state are updated but also the labels of those transitions).
- Context menus for most of the (graphical) components. This context menu features edit and delete functionalities.
- A dedicated editing dialog for most of these components.
- Multiple-component selections. These can be used to move or delete several components together.
- Save and load functionalities to and from an especially designed XML format (the Document Type Definition—DTD—of which can be found in appendix A)
- The possibility to export to the DiVinE format (.dve).

3.2 Modeling a system with Paxion

We will submit here an example of using our editor by modeling a somewhat peculiar system, while presenting screen-shots of the program as we progress. Note that in our editor, many actions can be achieved in several ways, but in order to keep this example short we will only present one way in the following text [7].

[7] If the reader plans to use the editor, he is invited to read the "HOWTO" file provided in the "release package" for some more (informal) explanations on how to use the program

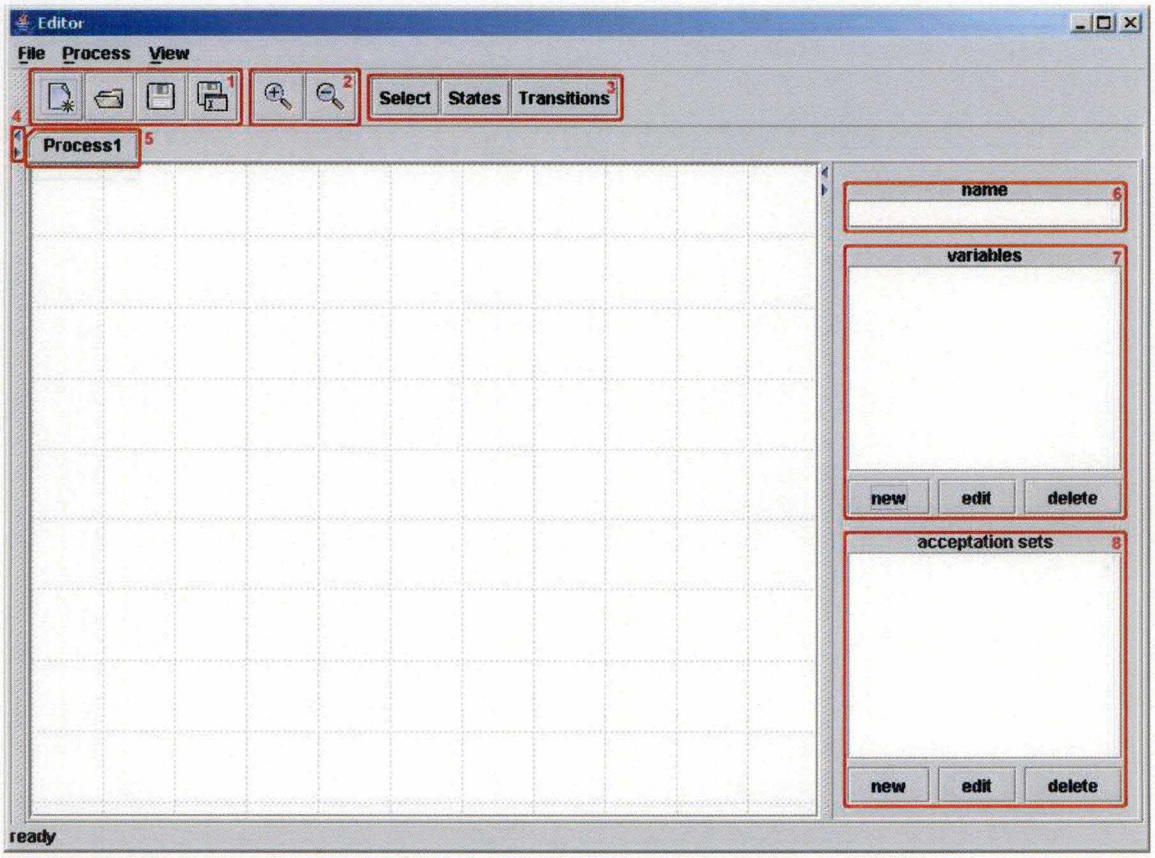

Figure 3.1: First screen of the program

First let us describe the figure 3.1 which shows a screen-shot of the program right after being started. We have highlighted and numbered some interesting zones on that screen-shot and we will refer to these while we explain.

The two first things one will probably notice are the *toolbar* and the "process properties panel" on the right.

The toolbar comprises icons for the usual file operations (new project, load project, save project, save project as...) **(1)**, for the *zooming* functionality **(2)** and for changing the *mode* of the "drawing area" **(3)**.

The last words above need a bit of explanation: the "drawing area" (the center "gridded" area in the screen-shot) has three modes: "Select mode", "State mode" and "Transition mode" (it starts in "State Mode"). Let us explain what one can do in each mode:

- Select mode: in this mode one can select, edit, move or delete any component (as long as the operation is supported by the component).
- State mode: this mode is mainly aimed at creating new states. But one can, additionally select, edit, move or delete existing states and their label.
- Transition mode: this mode is mainly aimed at creating new transitions. But, similarly to the "State mode", one can also select, edit or delete existing transi-

tions *and* select, edit, move or delete intermediate points and transition labels (guards, effects and synchronization labels).

Another fact we should mention is that the "system properties panel", which is positioned on the left of the drawing area, is hidden by default. Therefore, the first thing one should do if one wants to modify the system properties is to "unhide" the panel by clicking on the "java one click expand icon" on the top of the (visible) edge of the panel **(4)**.

Let us now let the user start modeling a system.

He started by setting the name of the "current" process in **(6)**. After he validated the new name by pressing the `ENTER` key, the title of the process tab **(5)** was updated automatically.

Then the user wished to set some properties for the system, thus he "unhided" the "system properties panel" (the "unhidden" panel is shown on figure 3.2).

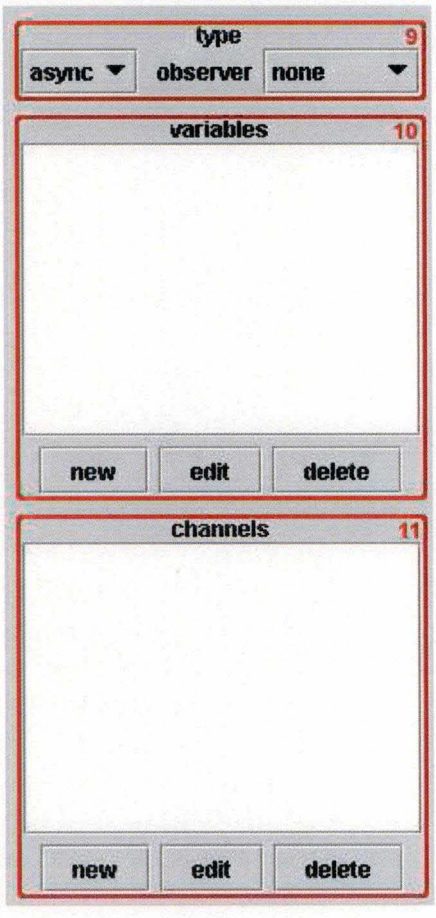

Figure 3.2: System properties panel

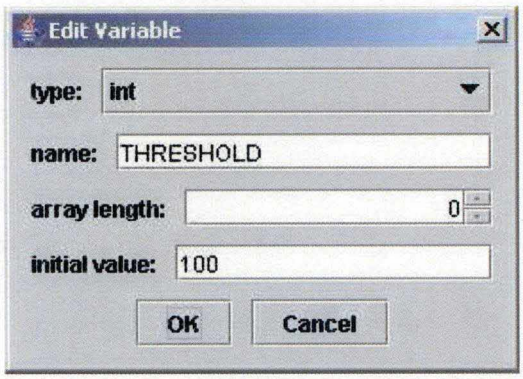

Figure 3.3: Variable editor dialog

The user, satisfied with the default type of the system (asynchronous), did not change it using the controls of the "type sub-panel" **(9)**.

But he did want to define a new global variable. For that he hit the "new" button of the "variables sub-panel" **(10)**, which brings up the "Edit variable" dialog

(figure 3.3). He defined a variable with an initial value and a length of 0 (which means it is not an array). In an analogous way, he defined new channels using the "new" button of the "channel sub-panel" **(11)**.

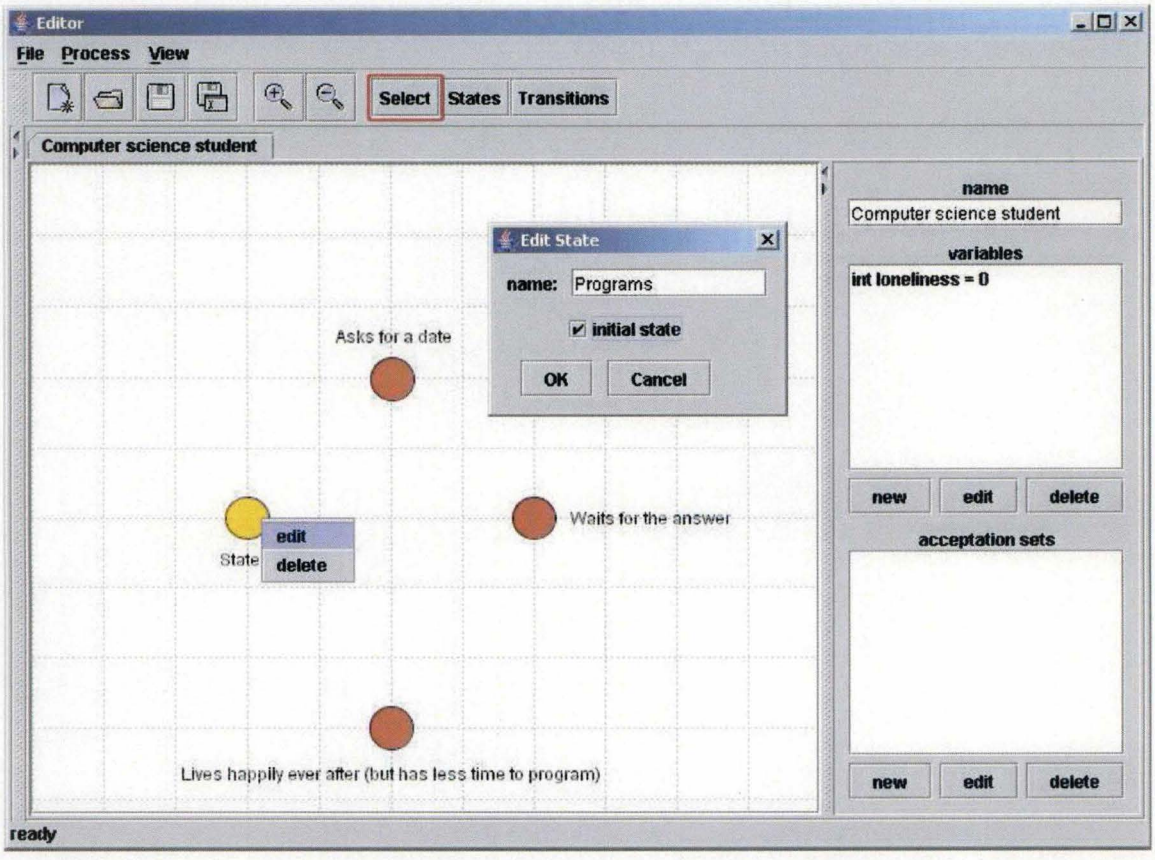

Figure 3.4: Editing a state using its context menu

The next step was to place the states. Our user first hided back the "system properties panel" to have more "screen space" to place the states. Since the drawing area was already in "State mode", the user could place states right away by simply clicking where he wanted to place them. After having placed the four states that his process comprises, he wished to change their name. This was done by using the "Edit State" dialog which is brought up by using the context menu of each state, as seen on figure 3.4. While editing the name of the last state, he took the opportunity to set it as the initial state for the process. Note that he decided to do that last step after switching to the "Select mode" and selecting the state (therefore the state being edited is orange but these are not required steps).

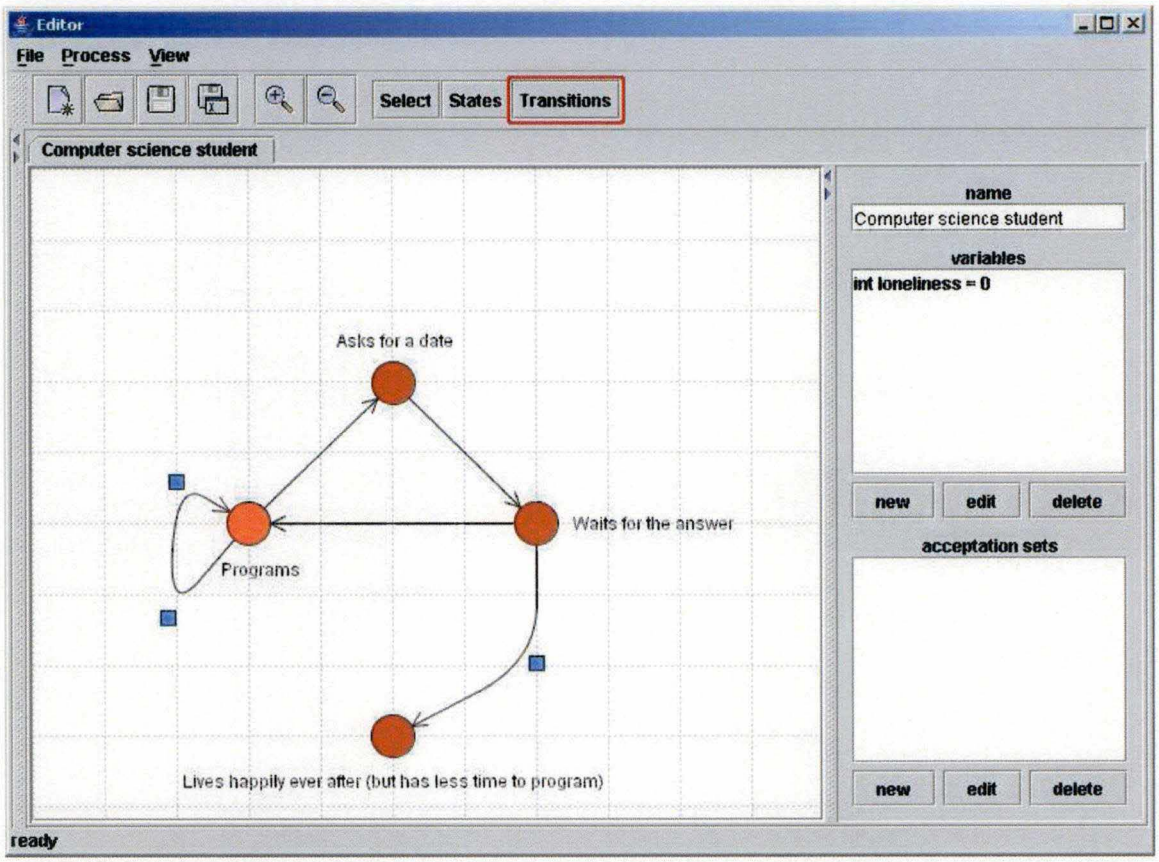

Figure 3.5: Placing transitions (including intermediates points)

Afterwards, he wished to define the transitions for that process. In order to do that, he first switched to the "Transition mode". All that was needed then was, for each transition, to click on the origin state of the transition then on its destination state. There are two things to notice on figure 3.5 besides the simple fact that the newly created transitions have appeared: first, the initial state defined in the previous step can be recognized by being brighter than the other states; secondly, two of the transitions he defined use intermediate points which are represented by small blue squares.

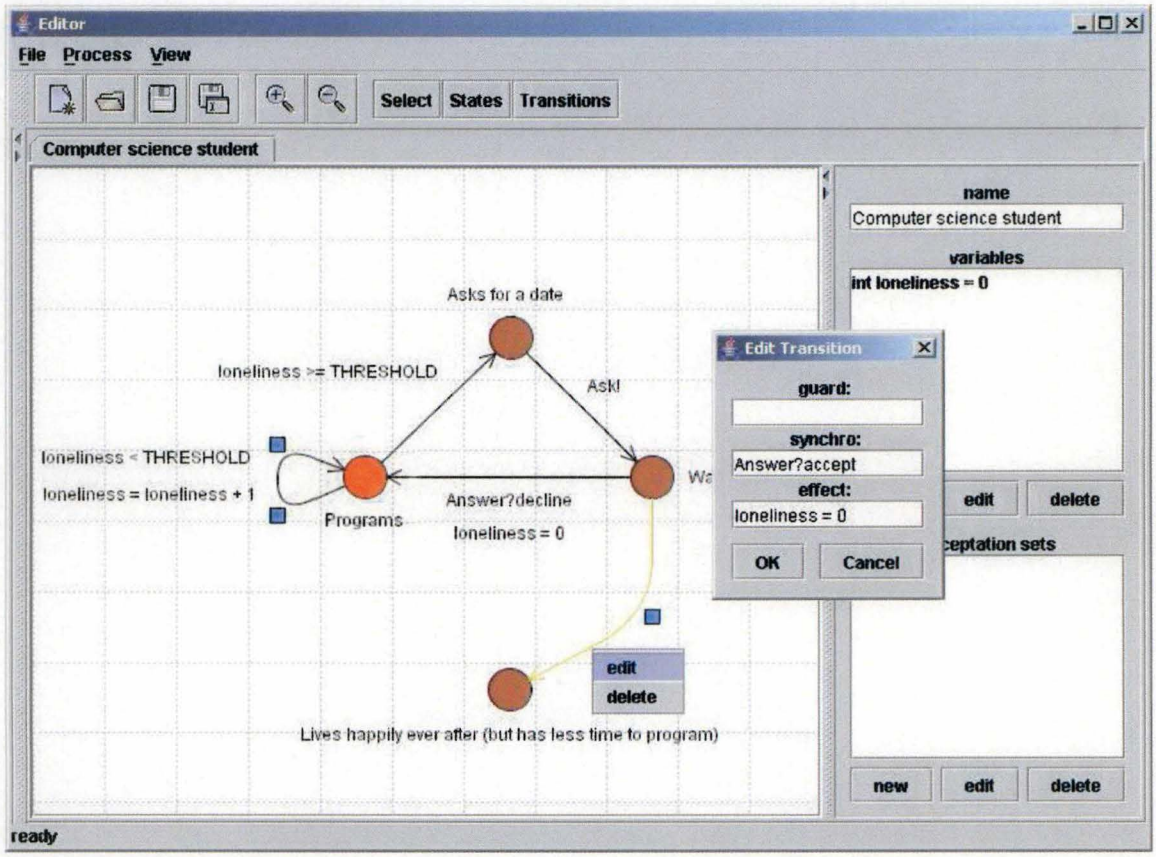

Figure 3.6: Editing the labels of a transition using its context menu

The user naturally wanted to edit the labels of the transitions of his process. He did so by using the "Edit Transition" dialog which was brought up using the context menu of each transition, as shown on figure 3.6.

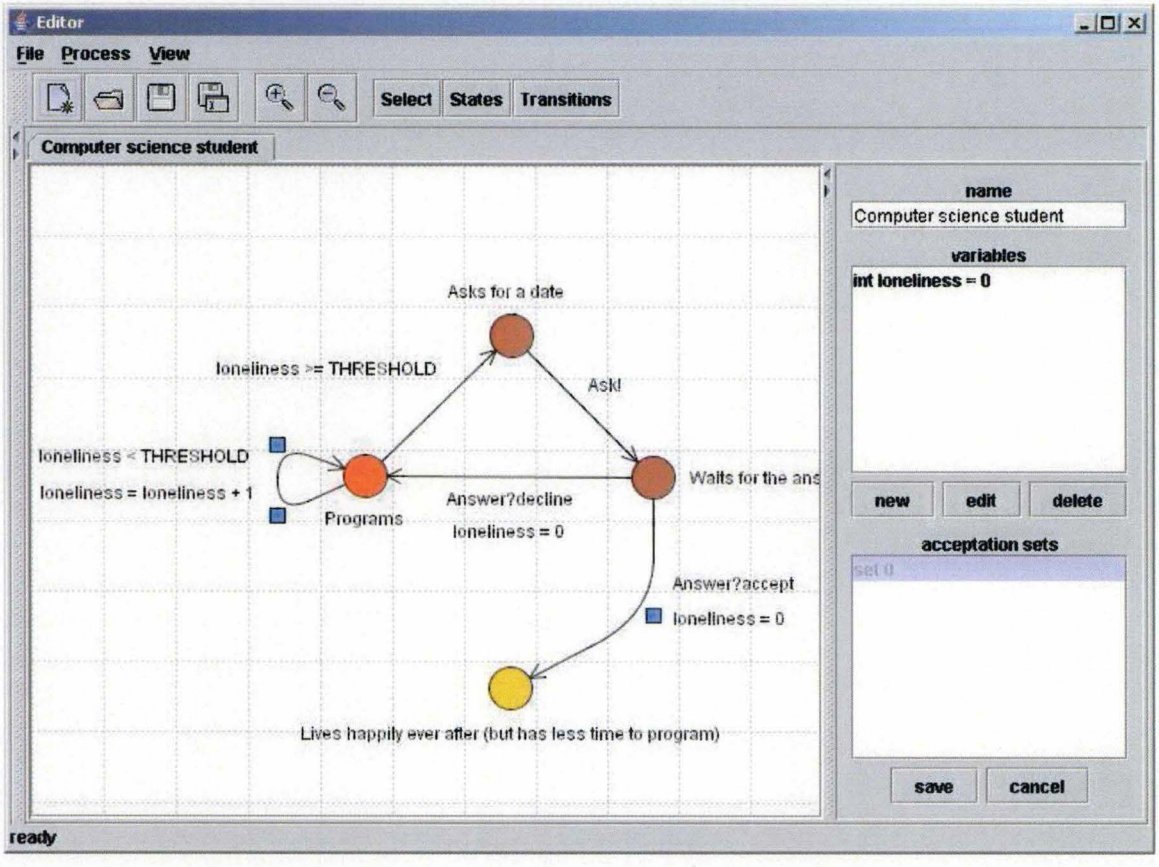

Figure 3.7: Editing an accepting set

Defining an accepting set is shown on figure 3.7. This action needs two steps. First, the user needed to create a new (empty) one by using the "new" button of the "acceptation sets" sub-panel (**(8)** on the first screen-shot) of the "process properties panel". Then, he could edit it by selecting the set in the list and click the "edit" button of the panel. He could finally add a particular state to that set (or rather "toggle" whether a state is part of the set) by clicking on it while holding the CTRL key. The states that are part of the *current* set are marked in orange as for a normal selection. An interesting fact to notice is that the buttons of the panel change to offer only the "save" and "cancel" functionalities while editing an accepting set.

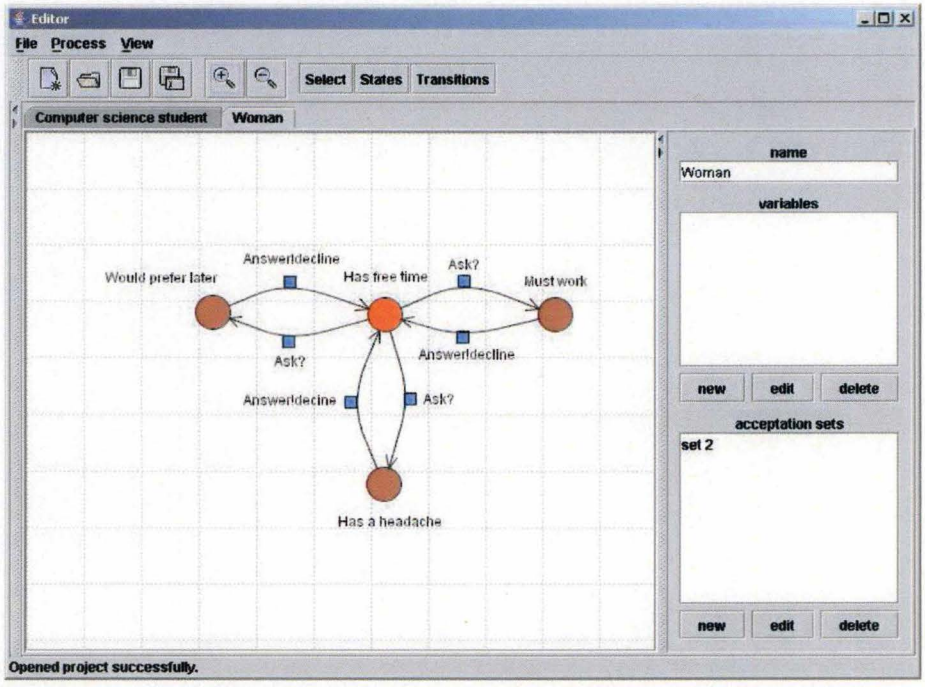

Figure 3.8: Editing another process

The figure 3.8 shows a later situation where the user had modeled another process for his system. And the last screen-shot on figure 3.9 simply shows that same process but "zoomed out". Additionnaly, on that screen-shot, both the "system properties panel" and the "process properties panel" are "open" at the same time.

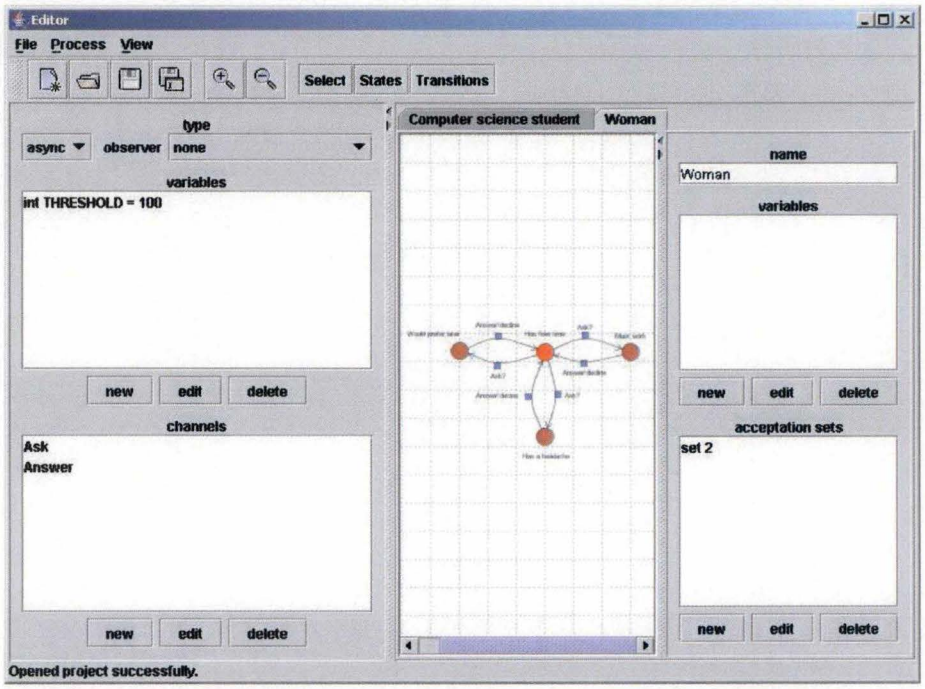

Figure 3.9: The system "zoomed out"

3.3 Architecture

In this section, we shall present the architecture of our program, including a description of every class. First, let us note that the classes (in the "object-oriented programming" meaning) of our program can be quite naturally divided into five categories: storage classes, (graphical) components, panels, dialogs, and miscellaneous classes. In addition to describing those categories of classes, we will also describe our interfaces (in the Java meaning of the term).

Note that we will try to be as concise as possible yet still describe every class. The reader who would like more information about interactions between classes is invited to look at the collaboration diagrams for the classes of each of the following sections (corresponding to the above-mentioned categories) in the appendix B.1 through B.5.

3.3.1 Storage classes

These classes are meant, as the name of the category suggests, to store data. Therefore they do not have any graphical representation.

A Project is an object representing a whole system. Such a system will usually consist of several *Processes* interacting with each other. The visual counterpart (and editor) of a *Project* object is a *ProjectEditor*, which will be detailed later.

A Process is an object to store the data concerning a process of a system (i.e. an automaton). It will often be part of a *Project* object. As for the *Project* object, the visual counterpart (and editor) of a *Process* object is a *ProcessEditor*.

A Channel is an object that represents a channel of a system. As we have seen in the first chapter of this work, these channels are used to communicate between processes.

A Variable is an object representing either a global (project) variable or a local (process) variable.

A VectorListModel is a container of objects meant to be used in a List. It was designed specifically as an alternative to Java's built-in *DefaultListModel* class, the biggest difference being that a VectorListModel implements the *Collection* interface. As the name of the class implies, the actual data is stored in a *Vector* object.

3.3.2 Graphical components

This is probably the most interesting category of classes. One reason of this is that this is the category which has the richest inheritance tree. Note that the first three classes mentioned below are not meant to be instantiated directly but rather have other classes inheriting from them.

AbstractComponent is an abstract class meant to represent objects that have a graphical representation and can be used on a *DrawingPanel*. This is the base class for all components of a *DrawingPanel*: all the graphical components inherit directly or indirectly from this class, as shown on figure 3.10.

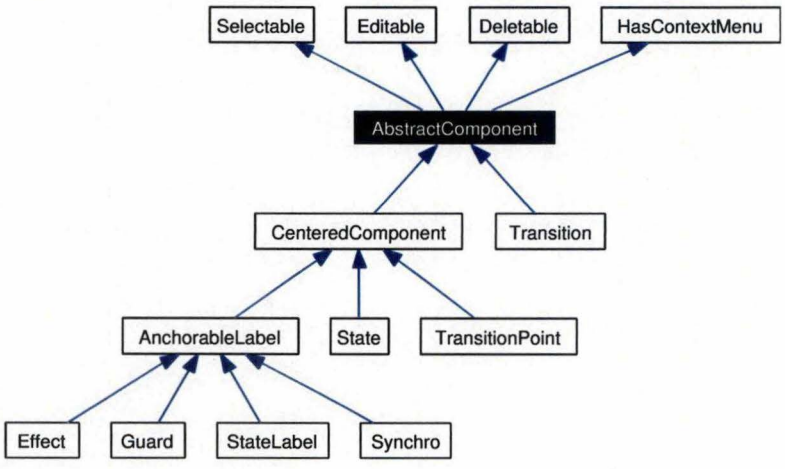

Figure 3.10: Inheritance diagram for the *AbstractComponent* class

A CenteredComponent is a component which extends an *AbstractComponent* by keeping track of its center point. The advantage of this, is that it can be moved by specifying a new position for its center. This is especially useful when components are interacted with using a mouse.

An AnchorableLabel is a class of components which extends the *CenteredComponent* class. It has the particularity to remember its position relatively to an external Point. We call that point the *AnchorPoint*. These *AnchorPoints* are used mainly for remembering the label's position relatively to its "parent" object (for example the *State* object of a *StateLabel*).

Instances of all the following classes serve both as data storage objects and as the graphical representation for them on a *DrawingPanel*. They can all "draw themselves" on a *DrawingPanel* they are added to.

A Guard, Effect or Synchro is an object that represents respectively the guard, effect or synchronisation label of a *Transition*.

A StateLabel is an object that represent a label of a State. The label of a State is, in fact, simply its name.

A State is an object representing one state of an automaton.

A Transition is an object representing one transition of an automaton. A transition can optionally have one *Guard*, one *Effect* and one *Synchro*nisation label or any combination of these. Additionaly it may also have one or several *TransitionPoint*(s) which are intermediate points for the arrow representing the transition.

As it is the case for the other classes described in this category, a *Transition* object serves both for the "graphical representation" of the transition on its *DrawingPanel*, and as the "data storage" object for it. Nevertheless it is peculiar in the fact that the actual drawing of the arrow representing the transition is delegated to an *Arrow* object.

A TransitionPoint represents a "draggable" intermediate point for a transition. Since it serves only to modify the path of a transition arrow, one could say a *TransitionPoint* has no real meaning: it is merely a visual artifact.

An Arrow is an object that represents a multiple-segment arrow. An Arrow object draws itself in a special way: it first draws half of its first segment with a straight line. Then, it draws a quadratic curve between the center of that segment (this corresponds to the end of the first half of the segment) and the center of the following segment, using the end point of the segment "of that iteration" as a quadratic parametric control point. It continues like this, iteratively, for all the following segments until the last one. That last segment has its second half drawn using a straight line.

3.3.3 Panels

An EditableListPanel is a generic panel to display and edit a list of *Editable* objects. The panel contains buttons to add, edit or delete such objects.

Note that the data itself (the list) is not local to this class and must be provided in the form of a *VectorListModel* object by the programmer who uses this class.

An AcceptStatesSetListPanel is an extension of an *EditableListPanel* meant to display more specifically a list of sets of accepting states.

A DrawingPanel is a panel which can be used to draw an automaton (representing a *Process*). This is certainly the most important class of our editor but unfortunately it is almost impossible to describe without entering into too much details. We will only say here that it serves as a container for components inheriting from the *AbstractComponent* class such as the *State* or *Transition* classes. It handles scaling (zooming) and takes care of all mouse events for the components it contains. We will further discuss that zooming feature later in this chapter.

A ProcessEditor is a panel designed to let the user edit a particular *Process*. As we have already said, the actual data is stored in the *Process* object and not the *ProcessEditor*. This panel contains mainly two things: a *DrawingPanel* (which does

most of the actual work) and a properties panel which serves to modify the process name, (local) variables and accepting states sets.

A ProjectEditor is to a *Project* what a *ProcessEditor* is to a *Process*. The *ProjectEditor* contains a panel where one can modify the properties of the project (set its type or declare variables and channels) and a tabbed panel with the different processes of the project (one tab per process).

3.3.4 Dialogs

An ObjectEditor is an abstract class providing a generic dialog which can be used to create specialized dialogs for editing the fields of simple objects.

The *ObjectEditor* provides "OK" and "Cancel" buttons only, the rest should be provided by subclasses. The "OK" button will update the edited object with the current values of the different dialog widgets and then close the dialog. The "Cancel" button will only close the dialog without updating the object.

In fact, all the dialog windows in our editor inherit from this class, as shown on figure 3.11

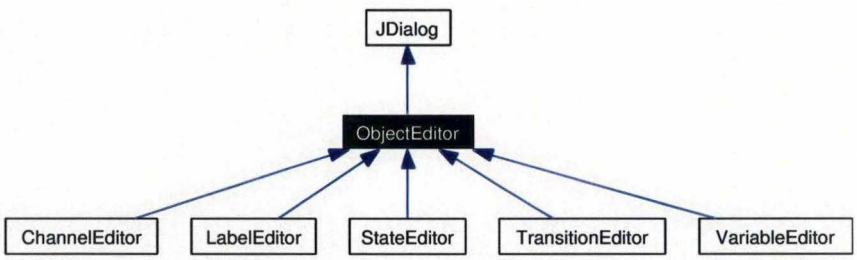

Figure 3.11: Inheritance diagram for the *ObjectEditor* class

The *ChannelEditor*, *LabelEditor*, *StateEditor*, *TransitionEditor* and *VariableEditor* classes each edit a particular *Channel*, *AnchorableLabel*, *State*, *Transition* or *Variable* respectively, but describing them further would be of little use.

A TransitionEditor is a simple dialog to edit a particular Transition. To edit transition means in fact to edit its labels (guard, effect, synchro) and this dialog reflects that, as seen on figure 3.6.

3.3.5 Miscellaneous

An Editor is the main object and the entry point into the program. It creates the menu bar, the tool bar and a *ProjectEditor* object to which it delegates the rest of the work. The *ProjectEditor* object is given a newly created (and therefore empty) *Project* to work on.

A ProjectParserDataHandler is an object used by the SAX parser to handle the XML entities it finds when it loads a project file. In other words this class is the heart of the "load from our XML format" functionality.

A CommonAction is an extension of Java's *AbstractAction*. The notable improvement is that it supports having a large icon in addition to the small one present in *AbstractAction*.

DivineFileFilter and XMLFileFilter classes are *FileFilter*s which accept directories and files with a ".dve" or ".xml" extension, respectively.

The Utils class is a special class which contains only static methods meant as "helper" methods for other classes. This class should not be instantiated.

3.3.6 Interfaces

In the current implementation, the only classes of components which can be used on a *DrawingPanel* are *AbstractComponent* and the classes derived from it. Since *AbstractComponent* implements the Deletable, Editable, HasContextMenu and Selectable interfaces, all components on a *DrawingPanel* object implement those interfaces.

The Deletable interface simply describes components that can be deleted from a *DrawingPanel*.

The Editable interface describes objects which know how to "edit themselves" (or rather how to launch an editor to edit them). Typically such an editor will be one of the dialogs presented above. The Editable interface is special, compared to the other interfaces described in this section, in the fact that it is the only one which is implemented by other classes than our graphical components (see its (partial) inheritance diagram on figure 3.12).

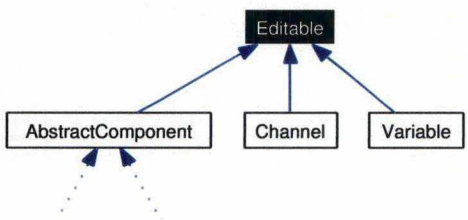

Figure 3.12: Partial inheritance diagram for the Editable interface

The HasContextMenu interface describes components from a *DrawingPanel* that have a ContextMenu.

The Selectable interface describes components that can be selected when they are used on a *DrawingPanel*. For a component, the state of being selected will typically involve a visual hint of it.

There is no restriction as to how the Selectable component can be selected, although in the current implementation all the ways of selecting a component involve using the mouse.

3.4 Implementation

3.4.1 Save and export functionalities

The "export to" the DiVinE format and the "save to" our own XML format functionalities share the same principle. We essentially provide these features "hierarchically". This means that each class is responsible to write its own information to the file which is being written, delegating to each of their "children" the task to write themselves and their children, etc.

For that to be possible, each of the Storage classes seen above and many of the Graphical component classes implement both a "writeXML" method and a "writeDivine" method. As an example of the implementation of one of these methods, we provide the code for the writeXML method of the *State* class in the listing 3.1.

Listing 3.1: the writeXML method of the *State* class

```
207     public void writeXML(PrintWriter writer) {
208         writer.println("<state id=\"" + id + "\">");

210         if (center != null)
211             writer.println("<point x=\"" + center.x + "\" y=\"" + center.y + "\"/>"
                    );

213         if (label != null)
214             label.writeXML(writer);

216         writer.println("</state>");
217     }
```

As examples of the output produced by these methods, we provide a partial listing in our XML format (listing 3.3) and the complete listing in the DiVinE format (listing 3.2) of a model of the Peterson Mutual exclusion protocol for two processes.

Listing 3.2: DiVinE file for the Peterson example

```
1  byte turn = 1, want1 = 0, want2 = 0;
2  process P_0 {
3  state CS, NC, wait, q1;
4  init NC;
5  trans
6  CS -> NC { effect want1=0; },
7  wait -> CS { guard want2 == 0 || turn == 2; },
8  q1 -> wait { effect turn = 1; },
9  NC -> q1 { effect want1 = 1; };
10 }

12 process P_1 {
```

```
13  state NC, wait, CS, q1;
14  init NC;
15  trans
16  q1 -> wait { effect turn = 2; },
17  NC -> q1 { effect want2 = 1; },
18  CS -> NC { effect want2 = 0; },
19  wait -> CS { guard want1 == 0 || turn == 1; };
20  }

22  system async;
```

Listing 3.3: (partial) XML file for the Peterson example

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE system SYSTEM "data/paxion.dtd">
3  <system type="0">
4  <variable type="1" name="turn" initialValue="1"/>
5  <variable type="1" name="want1" initialValue="0"/>
6  <variable type="1" name="want2" initialValue="0"/>
7  <process id="2" name="P_0">
8  <state id="8">
9  <point x="300" y="299"/>
10 <label text="wait">
11 <point x="38" y="1"/>
12 </label>
13 </state>
```

In addition to serving as an example of outputted "data" in this work, this example was originally modeled to assess whether our exported DiVinE format was valid. Indeed, the description used to model this example was, in fact, a file in the DiVinE format! We modeled it graphically using our editor, exported it to the DiVinE format and compared the two files. As expected, our generated file and the original file were semantically equivalent (yet not truly identical since they differ in the number and positions of line feeds and the grouping of the declaration of several variables on one line).

We also tested our export functionality by "feeding" several files exported from our editor to the simulator program comprised in DiVinE. We believe that, currently, our exported files are fully compatible with the format used by DiVinE.

3.4.2 Zoom

The zooming functionality was by far the most difficult and time-consuming feature to implement but also the most interesting from a programmer's point of view. We tried three different approaches in order to achieve it and only found the current one satisfactory. Let us briefly describe these approaches and the problems we encountered:

The first method that was tried was to use Java's built-in scaling feature[8]. We further wanted to have all our components inherit from Java's *JComponent* class (this is a basic class for graphical components), and to add those components in the standard way to a *JPanel* component (this is a Java class mainly meant at containing

[8]which has several advantages like, for example, the possibility for anti-aliasing (smooth the edges of) geometric figures

other graphical components). We will use hereafter the term "subcomponent" for components which are contained into another component.

An important fact to note about Java's built-in scaling feature is that it is visual *only*. This means that (most of) the coordinates used do not take the scaling into account. For example, when a component is scaled, the coordinates of mouse events it receives (or should receive) are not scaled accordingly (it receives the coordinates as if it was not scaled). Consequently, most of the very useful mechanisms built into the *JComponent* class fall apart when trying to scale a container component: the detection of whether some coordinates are inside the boundaries of a (sub)component or not, the dispatch of mouse events to the "correct" subcomponent and, even more importantly, the repaint algorithm, which is supposed to repaint everything which is modified.

We achieved a mostly-working implementation with this method but this involved using many "side tricks". But we eventually ran into a problem that we were sure we could not solve using this approach. It is too complicated to explain in detail here but let us just state that the reason we could not solve the problem was that it involved some internal Java methods over which we have no control.

The second method was to do the scaling (zooming) of the subcomponents by hand but still, as for the first method, have them inherit from the *JComponent* class. At that point, all our graphical components included a *scaling* variable and when their scaling changed, they had to have their bounds manually updated (i.e. position and size) and had to call the scaling method of components "linked to" them. It was less visually appealing than the first method and ran into some of the same problems as the first approach and therefore was abandoned too.

The third and current method is to use Java's built-in scaling feature as in the first approach but really only have one *JComponent* object: the container (this class is called DrawingPanel in our implementation), all the subcomponents being "custom" classes. This approach has several drawbacks but also many advantages. The main drawback is that we loose all the built-in features of the *JComponent* class for the subcomponents, therefore we had to re-implement manually many of these. The main advantage is that one has more control on these custom classes, which leaves much more room for "custom optimization" (even though this has not been done yet).

This concludes this chapter which was core to this paper. After having submitted a detailed description of Paxion and an overview of its implementation, we shall now cast a critical eye on what we have achieved.

Chapter 4

Evaluation and Improvements

In this fourth and last chapter we will discuss the perspectives of improvements for our editor. Some of these improvements are just correcting errors while many others correct shortcomings which are due to having been out of our original scope. This chapter will be divided into several sections:

Since the possibilities of improvements and extensions are plentiful, we will only briefly outline some of them.

One should also note that there is little distinction between what we may have done wrong and the desired extensions. We will thus often discuss "how we could extend the editor to overcome its weaknesses".

We have [[classified]] these improvements into XX categories: *visual formalisms*, *import* functionalities, *export* functionalities, improved *interaction* with the backend,

Our main error was to implement our editor from scratch instead of extending an existing editor or at least one of the libraries presented in the second chapter.

Let us now further categorize our analysis: we will subdivide it into: "visualization and user interface", "architecture" and "implementation".

4.1 Visualization and user interface

4.1.1 Modeling formalism

We have already stated that the current visual formalism is not very well suited for very large systems. In order to support another or an extended formalism, there are two possibilities:

Either, rely on our "backend" to support the "extended" formalism. In that case, we believe supporting a formalism like the statecharts would not be as hard as one could imagine. Concerning the hierarchy of states/components in itself, the first step would be to generalize our AnchorableLabel to some kind of AnchorableComponent with a parent Component to have a "draggable" hierarchy of component. Of course, that would not be the only required feature but that is a good start.

In case the "backend" does not support the "extended" formalism, the editor

could still provide the back-end with models in its own formalism, on the condition that a converter from one formalism to another is feasible. But this approach involves more difficulties than the first one.

4.1.2 Observer Automaton

To consider the observer automaton of a system like just another process was probably inappropriate. It would have been better if the observer process was more independent from the system. Indeed, one will usually want to verify several properties for a single system. Assuming one wants to specify those properties using observer automata, one would have to save the same system multiple times (for each of the different observer automata) in order to do so.

Letting the user define multiple observer processes for a single system and then letting him choose which one he wants to use for the export to DiVinE would probably be a good answer to the above.

4.1.3 Accepting sets

Our handling of accepting sets (of states) is highly un-intuitive. The most obvious shortcoming of which is probably the fact we do not allow a permanent representation of these sets. Unlike other shortcomings, this was not an oversight.

The problem was that, at the time of the implementation, we had no idea on how to represent them. This can be easily solved for finite state machines by adding a (smaller) circle inside the accepting states. But, for generalized Büchi automata, there are two additional problems:

- It is possible to have several different sets of accepting states, so the editor needs to provide the user with a way to differentiate those.
- One state can be part of several accepting sets[1].

The first problem can be quite easily overcome, for example, by using different colors for the sets. But we had not found how to resolve the second problem until recently: one could simply use concentric circles of different colors (either outside or inside the boundary of the state) to represent the different accepting sets of one state. There is still the small problem of what to do if one state is part of many different accepting sets. An elegant way to solve this would be to reserve one specific color within the possible colors for our concentric circles which would mean "this state is part of more accepting sets than the threshold". The full list of accepting sets could still be made available for example on the "Edit state" dialog.

[1] in practice we do not know if this situation appears often or at all but there is nothing preventing that in the definition of Generalized Büchi automata

4.1.4 Properties panels

Our "double side bar" approach (one side bar for the "process properties panel" and one for the "system properties panel") is quite unpractical (it uses so much screen space that it forced us to hide the "system properties panel" by default to have some space to draw the actual automaton).

4.2 Architecture

4.2.1 Packages are under-utilized

In order to implement the architecture depicted above, the classes could have been separated into different packages. At the moment Paxion is composed of only one single package. The partition we used in section 3.3 to describe the classes (components, data, dialog, panel and misc(ellaneous)) appears to be reasonable (or clean) on first sight. But having the DrawingPanel separated from the (graphical) components it contains would probably cause to declare many more public methods than necessary, which is considered "unclean" by many.

4.2.2 Properties panels

Properties panels of the ProjectEditor and ProcessEditor classes could have been separated in their own classes to allow for more re-usability and also to simply reduce the size of these two classes.

4.3 Implementation

4.3.1 Save and export functionalities

As we have seen in section 3.4.1, the code for saving to XML and exporting to the DiVinE format is scattered to many classes: each class writing its own bit of information into the destination file. While this is a natural approach in an object-oriented program (everything that concerns a class of objects is gathered in that class), it has an important drawback: if ones want to add an export functionality for some new format, one has to add new methods in many classes.

Another approach (which would facilitate the writing of new exporting functionalities) would be to write a specific class for exporting a system for each particular format. These classes would have to fetch the information they need in all the specific objects. The amount of code would be roughly the same as it is now but, to add a new format, one would only have to create one file (for the new class) and not have to effect modifications in many different classes.

4.3.2 Import functionalities

We have already stated in the previous chapter that we do not support any importing functionality, but, as we have also said, it would be an interesting feature from a user point of view. Providing such a functionality mainly involves two things: to "understand" the language being imported (i.e. have a parser for it); and to be able to automatically layout the imported elements.

One approach to facilitate our work would be to provide the functionality for only one language (for example, the DiVinE language) and then rely on external tools to convert from other languages to that language. In order to implement such a feature, it would probably be a good idea to use, or at least get inspiration from the libraries providing automatic layout that we presented at the end of the second chapter.

4.3.3 Component display speed

We do not think the speed is a critical issue in a graphical editor. Nevertheless it should be able to edit fairly large systems (i.e. with many states) without noticeable slowdown. In its current state, our editor does slow down for large systems. The speed could be greatly improved by several methods:

- First, let us note that, in the current state, one of the slow elements of the "drawing process" is the drawing of the grid. It would probably speed up the process to "cache" the grid, thus not redraw it as often as is currently the case.

- One more difficult approach, but which would probably result in a much greater speed improvement would be to store the states using a special "two dimensional sort algorithm". This would allow us to only "ask" the states which are currently visible to draw themselves instead of "asking" all of them and then relying on a clip rectangle[2] as is currently the case. Note that this kind of optimization is probably only possible in the "custom components" approach, as described in the end of the third chapter.

4.3.4 Improved interaction with the back-end

Currently the interaction with the "back-end" is only done through exporting to its format, which is quite limited. Our editor could greatly benefit from a more direct and much more adavnced interaction.

Examples of interactions an editor could have:

Simulation, but we think one should not limit oneself to the two features that come to mind immediately: "step-by-step" simulation and trace gathering and visualization. We think such a simulator should try to duplicate the functionalities of currently used debuggers:

[2] a clip rectangle is a rectangle which delimits the currently "drawable" area. Everything which is outside the rectangle is "clipped" and thus not drawn.

- step-by-step execution, but also, if the formalism is hierarchical:
  - step in : advance one step and possibly go into a "more detailed" component
  - step out : advance one step and get out of the current component
  - step over : advance one step but do not enter into a possibly "more detailed" component
- run until one hits a particular state, or even better: add breakpoints on particular states.

In the case of our editor, an integration with an LTL to Büchi automata converter (as exists in DiVinE) could also be quite a good feature. The best situation would be to be able to specify a property in LTL within the editor, and then to be able to visualize and possibly edit the observer process corresponding to that formula.

Lastly, to be able to graphically visualize a trace of events leading to a non verified property would probably help the user to find the problem quicker in his system.

Conclusion

In this paper we have presented both an overview of what model checking is, and the basics of how one can perform LTL model checking. We have then elaborated upon how one could visualize more complex systems. And, after a short summary of the existing graphical editors for finite-state-machine-like formalisms, we have submitted our own editor, its limitations and how it could be extended.

As we have seen, it may have been a mistake to create an editor from scratch. It might probably have been better to adapt an existing graphical editor (possibly from another domain) or at least to use more libraries to facilitate the implementation.

Also, elaborating on what was said in the fourth chapter, we can affirm that, in its current state, our editor is still far from being a "commercial-quality" product (although we are not displeased with it). It is also probably not practical to use for modeling very large systems as is often needed in the industry. But, as noted before, extending it to support hierarchy would not be extremely hard, provided the underlying "back-end" supports it. In its current version, our editor is probably only useful for modeling small systems for demonstration or teaching purposes *and*, of course, for possible extensions.

As far as the purpose of providing DiVinE with a graphical editor (or, more generally speaking a graphical user interface) is concerned, the choice is now open : to extend, modify or at least re-use parts of our editor, to adapt another editor, or to start a new editor from scratch. The last possibility is the only one we can qualify with a high degree of confidence as useless. In the current situation, we strongly feel that extending our editor would be the best choice (i.e. the choice representing the least amount of effort in order to achieve a "commercial-quality" product).

We have provided in the last chapter several directions in which our editor could be extended. It would be interesting to see these extensions actually implemented. Having said that, we sincerely hope our work will indeed be used and extended.

Bibliography

[1] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, Ph. Schnoebelen, and P. McKenzie. *Systems and Software Verification: Model-Checking Techniques and Tools.* Springer-Verlag, Berlin; Heidelberg; New York; ..., 2001. ISBN : 3-540-41523-8.

[2] N. Dulay. *Darwin Tutorial & Reference.*

[3] David Harel. *On visual formalisms.* Communications of the ACM, 31(5):514–530, 1988.

[4] ParaDiSe laboratory. *Networks of Extended Finite State Machines: Syntax, Semantics, and Examples* - http://anna.fi.muni.cz/ddt/semantics.ps.

[5] S. Mauw. *The formalization of Message Sequence Charts.*

[6] Sun Microsystems. *How to Write Doc Comments for the Javadoc Tool* - http://java.sun.com/j2se/javadoc/writingdoccomments/.

[7] David Safranek. *Graphical Specification of Concurrent Systems* - Ph.D. Thesis Proposal.

[8] Pierre Wolper. *The Algorithmic Verification of Reactive Systems* - Slides from the 1998 Francqui Chair lectures given at the FUNDP (Namur).

Appendix A

DTD

Listing A.1: DTD

```
<!ELEMENT system (channel*, variable*, process*, observer?)>
<!ATTLIST system type CDATA #REQUIRED>

<!ELEMENT channel EMPTY>
<!ATTLIST channel name CDATA #REQUIRED>

<!ELEMENT variable EMPTY>
<!ATTLIST variable
    name CDATA #REQUIRED
    type CDATA #REQUIRED
    length CDATA #IMPLIED
    initialValue CDATA #IMPLIED>

<!ELEMENT process (variable*, state*, initstate?, acceptset*, transition*)>
<!ATTLIST process
    id CDATA #REQUIRED
    name CDATA #REQUIRED>

<!ELEMENT observer EMPTY>
<!ATTLIST observer id CDATA #REQUIRED>

<!ELEMENT state (point?, label?)>
<!ATTLIST state id CDATA #REQUIRED>

<!ELEMENT initstate EMPTY>
<!ATTLIST initstate id CDATA #REQUIRED>

<!ELEMENT acceptset (#PCDATA)>
<!ATTLIST acceptset id CDATA #REQUIRED>

<!ELEMENT transition (point*, guard?, synchro?, effect?)>
<!ATTLIST transition
    from CDATA #REQUIRED
    to CDATA #REQUIRED>

<!ELEMENT point EMPTY>
<!ATTLIST point
    x CDATA #REQUIRED
    y CDATA #REQUIRED>

<!ELEMENT label (point?)>
<!ATTLIST label text CDATA #REQUIRED>

<!ELEMENT guard (point?)>
<!ATTLIST guard text CDATA #REQUIRED>
```

```
<!ELEMENT synchro (point?)>
<!ATTLIST synchro text CDATA #REQUIRED>

<!ELEMENT effect (point?)>
<!ATTLIST effect text CDATA #REQUIRED>
```

Appendix B

Collaboration diagrams

B.1 Dialogs classes

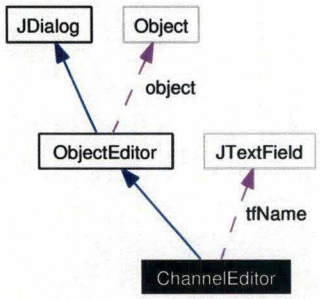

Figure B.1: Collaboration diagram for ChannelEditor

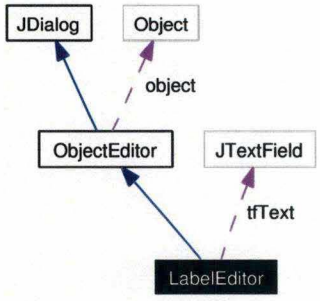

Figure B.2: Collaboration diagram for LabelEditor

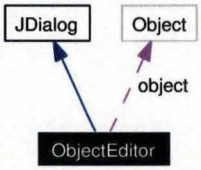

Figure B.3: Collaboration diagram for ObjectEditor

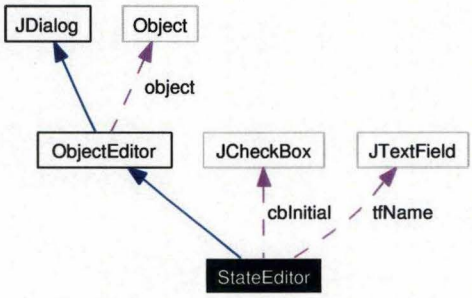

Figure B.4: Collaboration diagram for StateEditor

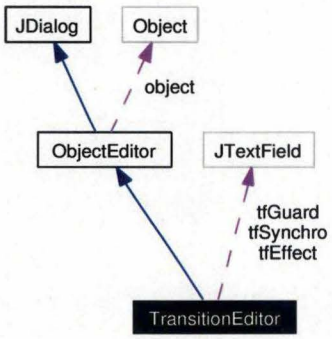

Figure B.5: Collaboration diagram for TransitionEditor

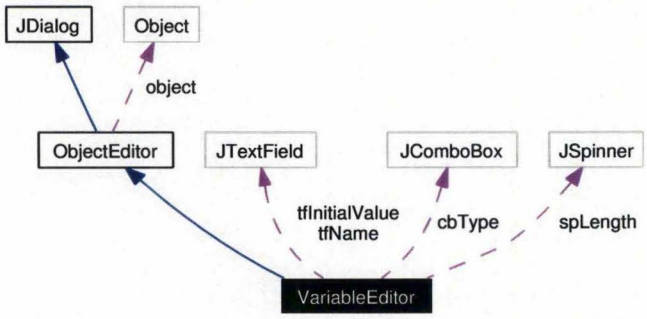

Figure B.6: Collaboration diagram for VariableEditor

B.2 Graphical component classes

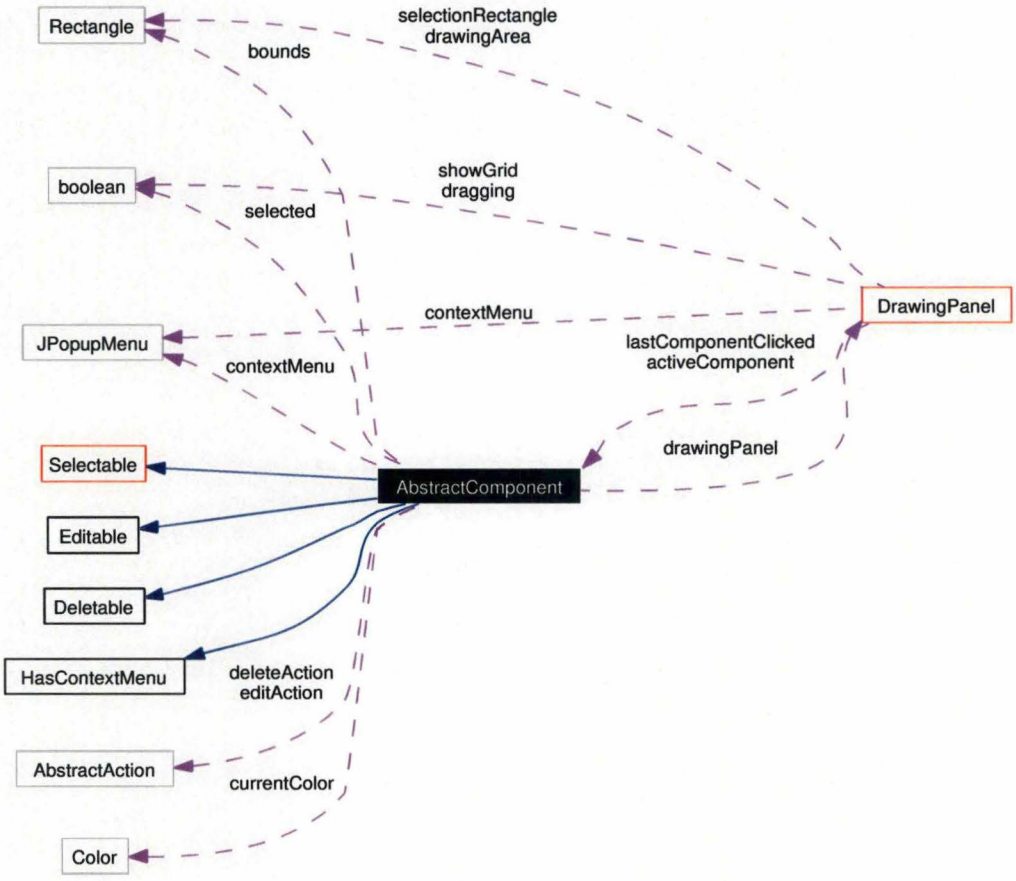

Figure B.7: Collaboration diagram for AbstractComponent

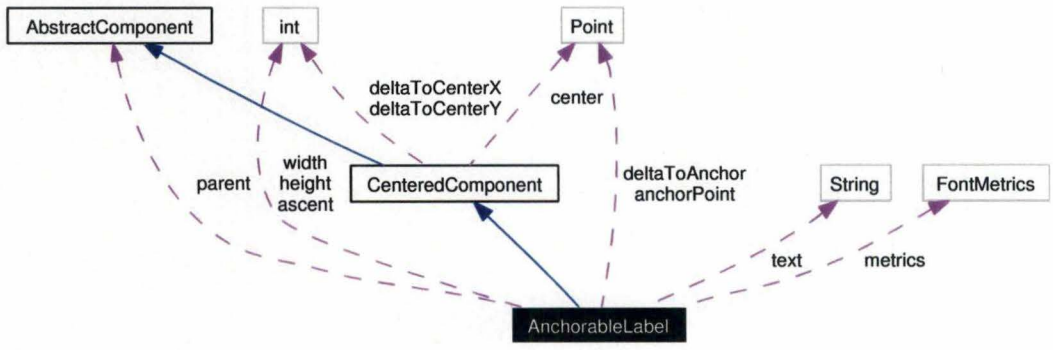

Figure B.8: Collaboration diagram for AnchorableLabel

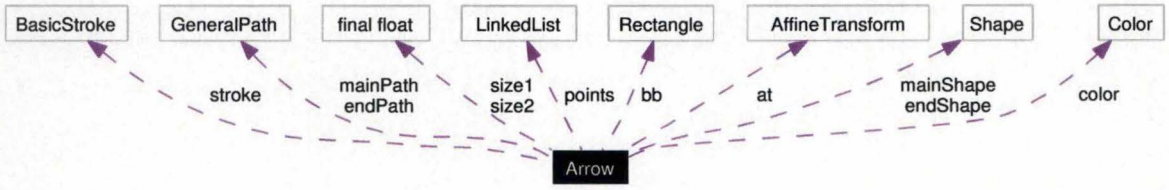

Figure B.9: Collaboration diagram for Arrow

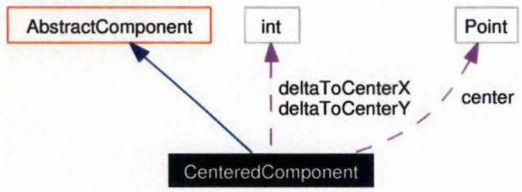

Figure B.10: Collaboration diagram for CenteredComponent

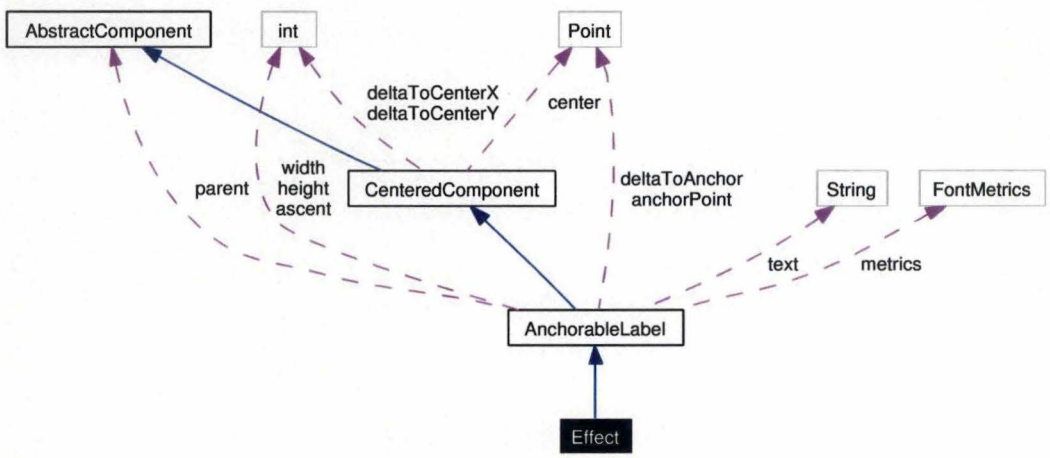

Figure B.11: Collaboration diagram for Effect

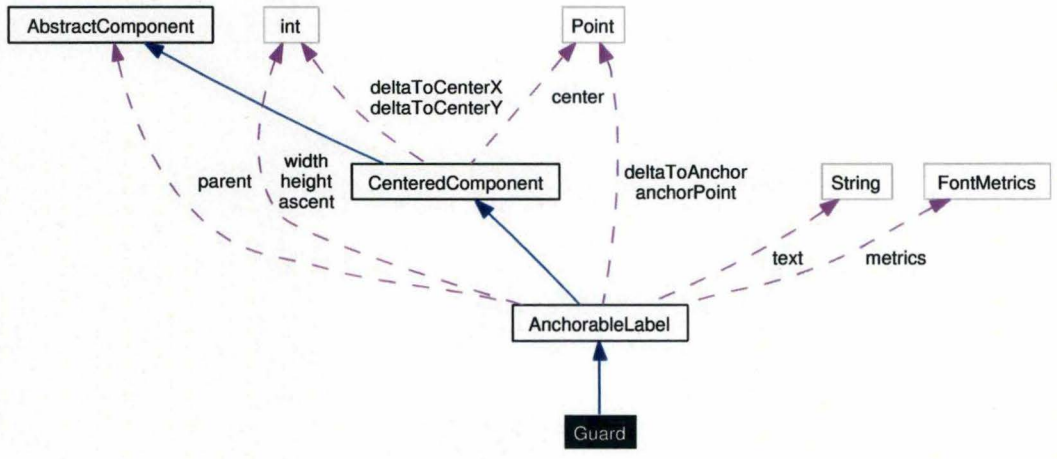

Figure B.12: Collaboration diagram for Guard

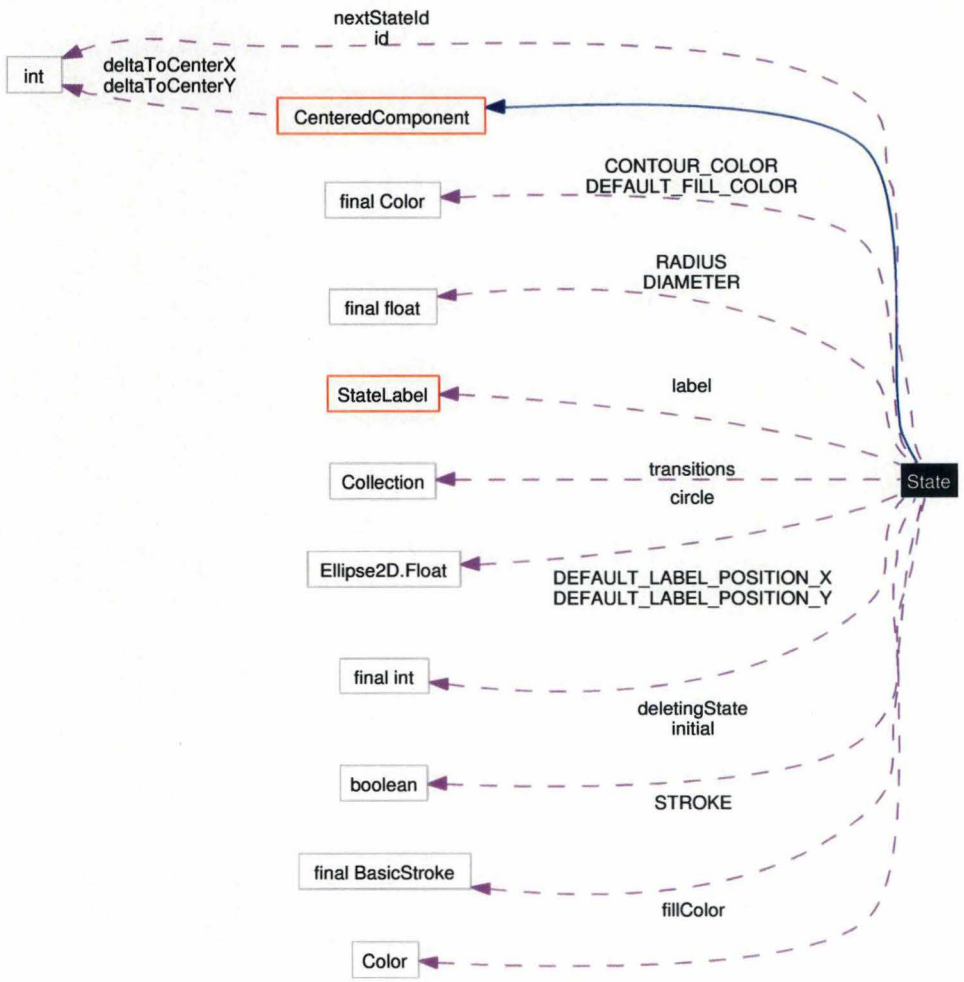

Figure B.13: Collaboration diagram for State

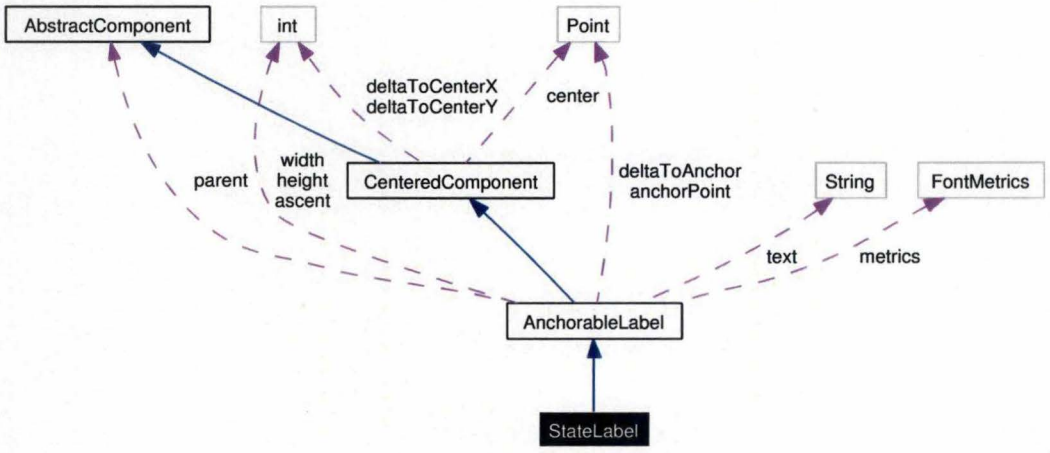

Figure B.14: Collaboration diagram for StateLabel

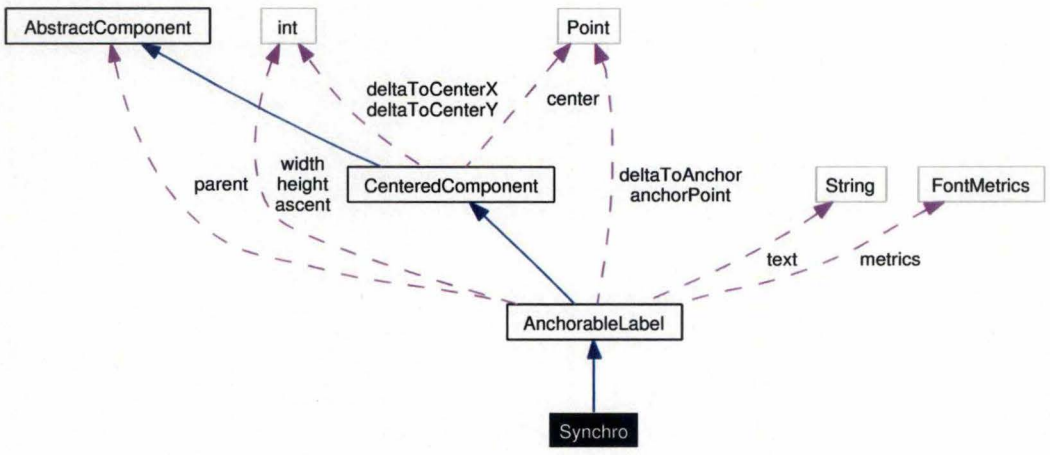

Figure B.15: Collaboration diagram for Synchro

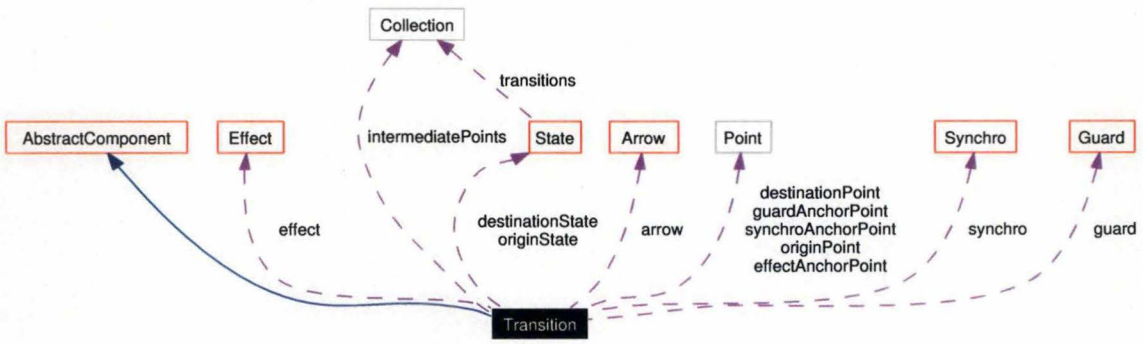

Figure B.16: Collaboration diagram for Transition

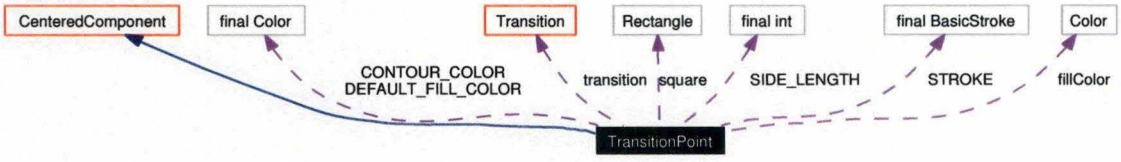

Figure B.17: Collaboration diagram for TransitionPoint

B.3 Miscellaneous classes

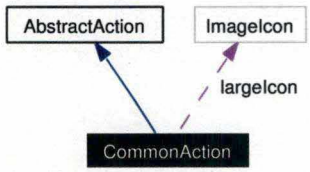

Figure B.18: Collaboration diagram for CommonAction

Figure B.19: Collaboration diagram for DivineFileFilter

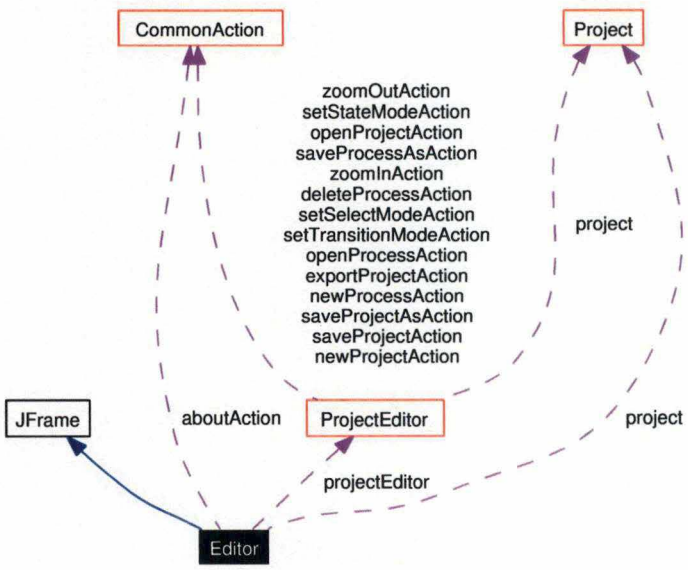

Figure B.20: Collaboration diagram for Editor

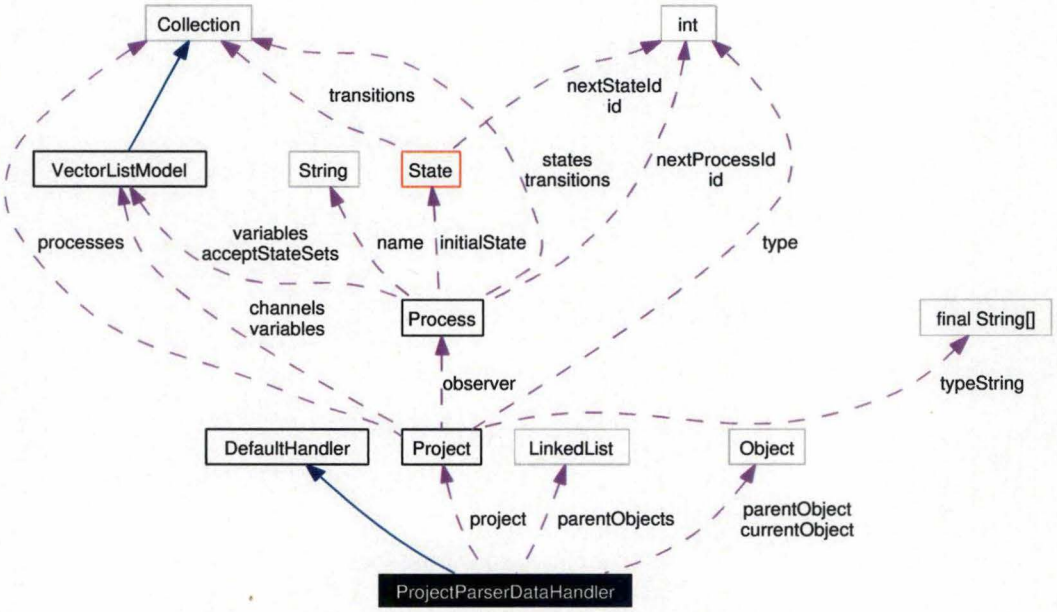

Figure B.21: Collaboration diagram for ProjectParserDataHandler

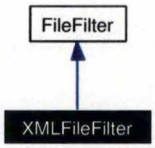

Figure B.22: Collaboration diagram for XMLFileFilter

B.4 Panel classes

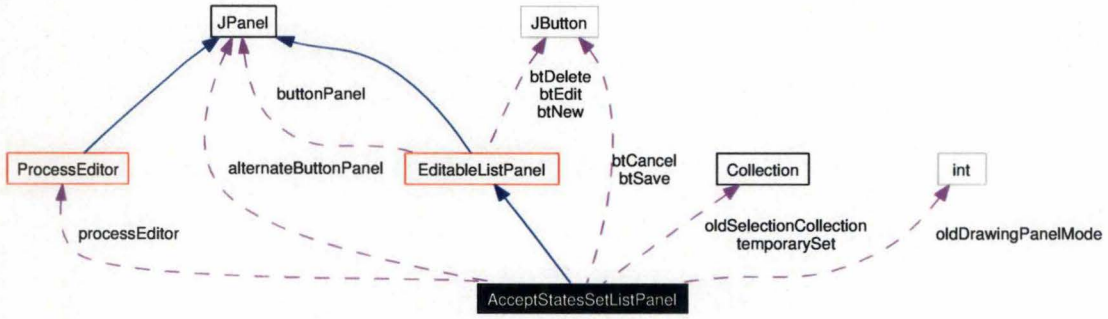

Figure B.23: Collaboration diagram for AcceptStatesSetListPanel

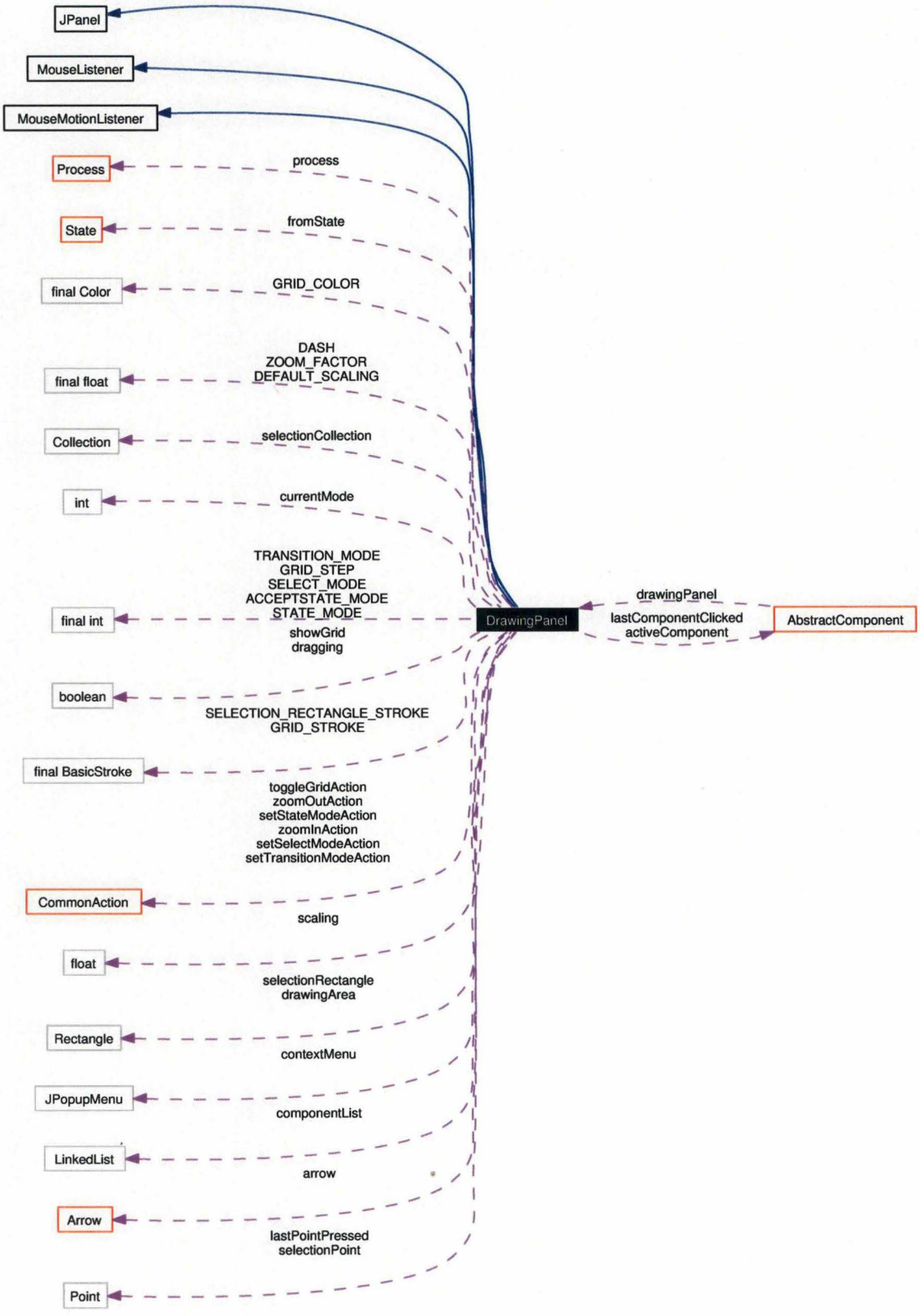

Figure B.24: Collaboration diagram for DrawingPanel

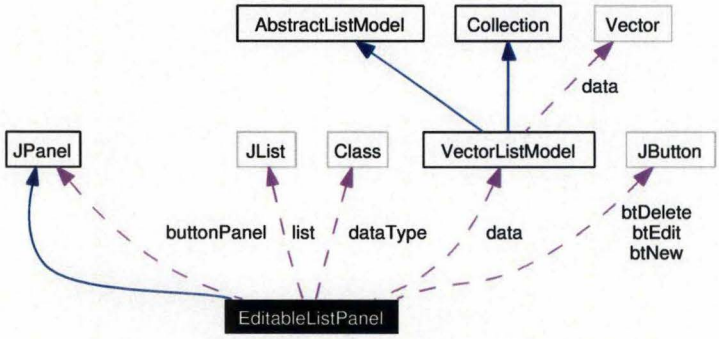

Figure B.25: Collaboration diagram for EditableListPanel

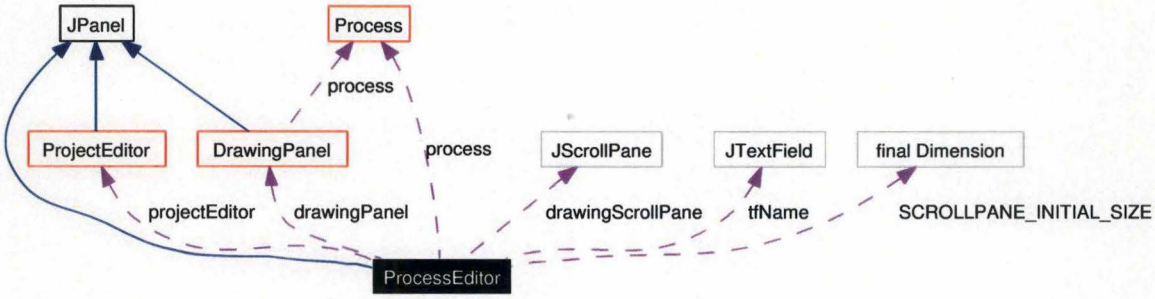

Figure B.26: Collaboration diagram for ProcessEditor

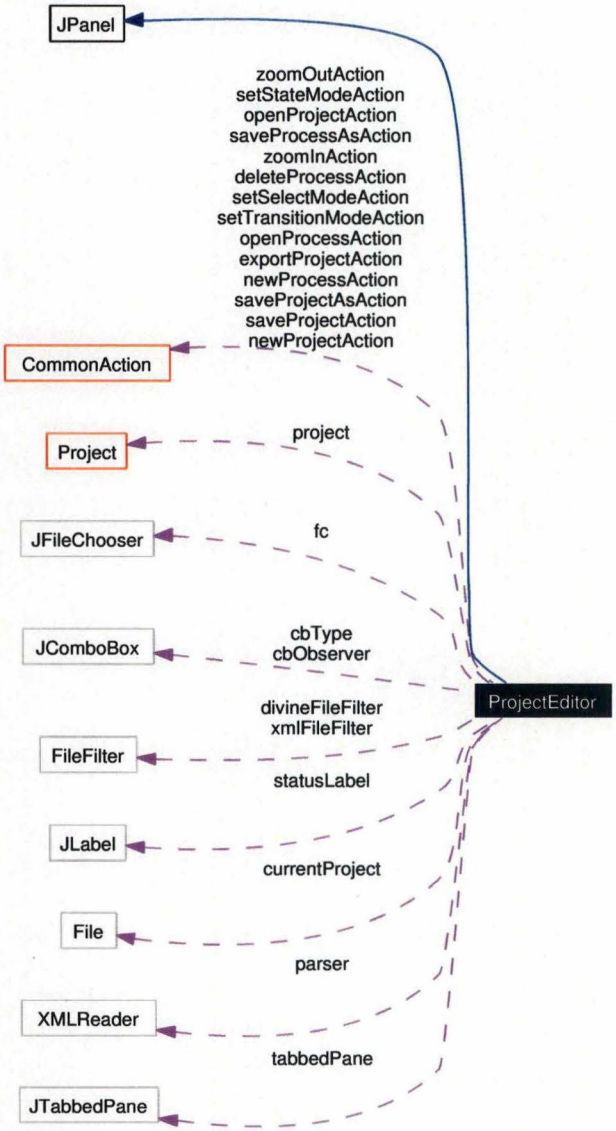

Figure B.27: Collaboration diagram for ProjectEditor

B.5 Storage classes

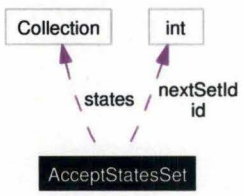

Figure B.28: Collaboration diagram for AcceptStatesSet

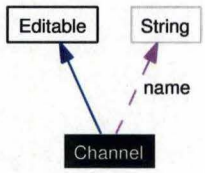

Figure B.29: Collaboration diagram for Channel

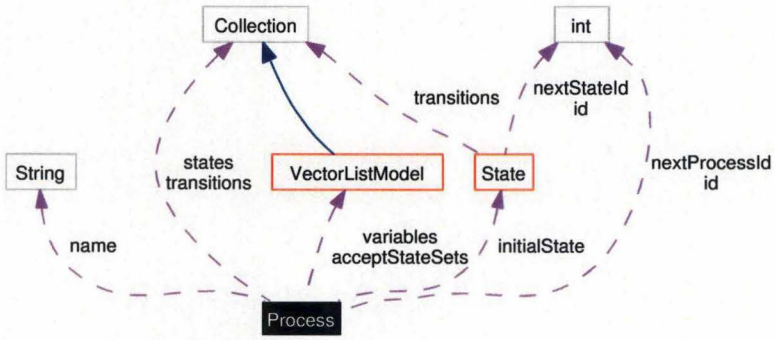

Figure B.30: Collaboration diagram for Process

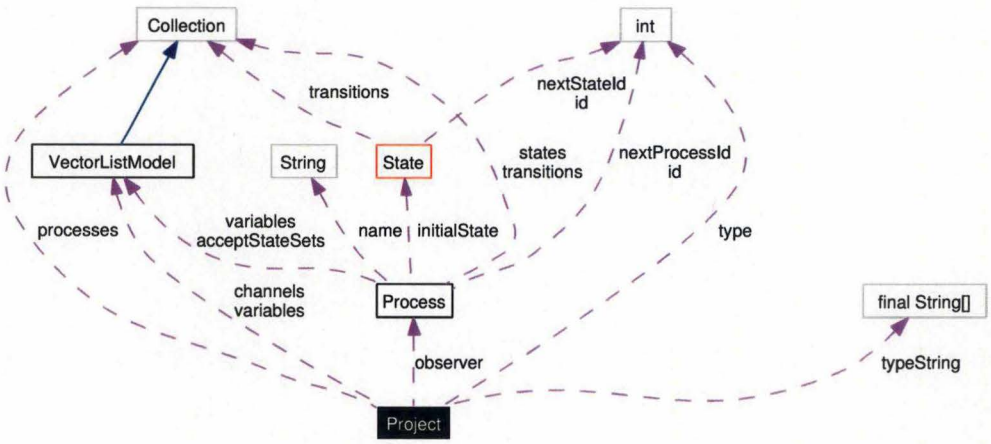

Figure B.31: Collaboration diagram for Project

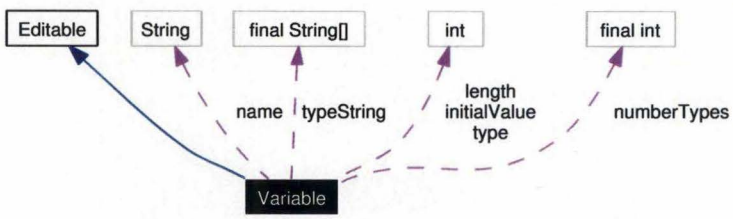

Figure B.32: Collaboration diagram for Variable

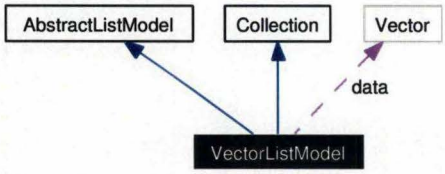

Figure B.33: Collaboration diagram for VectorListModel

Appendix C

Source code

C.1 AbstractComponent

Listing C.1: Source code for AbstractComponent

```
package paxion;

import java.awt.*;
import java.awt.event.*;

import javax.swing.AbstractAction;
import javax.swing.JPopupMenu;

public abstract class AbstractComponent implements Selectable, Editable, Deletable,
     HasContextMenu {
    //—— FIELDS ——

    protected Color currentColor = DEFAULT_NORMAL_COLOR;
    protected DrawingPanel drawingPanel = null;

    protected JPopupMenu contextMenu;

    protected AbstractAction editAction = new AbstractAction("edit") {
        public void actionPerformed(ActionEvent e) {
            edit();
        }
    };

    protected AbstractAction deleteAction = new AbstractAction("delete") {
        public void actionPerformed(ActionEvent e) {
            delete();

            // This must be done here and not in the delete method itself to avoid
                a
            // ConcurrentModificationException (on the currentSelection HashSet)
            // when the delete method is called by deleteSelected from DrawingPanel
            drawingPanel.removeFromSelection(AbstractComponent.this);
        }
    };

    // bounds of the component
    protected Rectangle bounds = new Rectangle();

    private boolean selected = false;

    /**
     * Constructs a new AbstractComponent.
```

```
     */
    public AbstractComponent() {
        contextMenu = createContextMenu(getContextMenuActions());
    }

    //------ PUBLIC METHODS ------

    /**
     * Returns the color used to paint the component when it is selected.
       Components
     * inheriting from <code>AbstractComponent</code> should override this method
       if they want
     * to use another color when they are selected.
     *
     * @return the color used to paint the component when it is selected
     */

    /* It would be sensible to have this method static but since subclasses can't
       override
     * static methods, we have to use a non-static one.
     */
    public Color getSelectedColor() {
        return DEFAULT_SELECTED_COLOR;
    }

    /**
     * Returns the color used to paint the component when it is not selected.
       Components
     * inheriting from <code>AbstractComponent</code> should override this method
       if they
     * want to use another color when they are not selected.
     *
     * @return the color used to paint the component when it is not selected
     */

    /* It would be sensible to have this method static but since subclasses can't
       override
     * static methods, we have to use a non-static one.
     */
    public Color getNormalColor() {
        return DEFAULT_NORMAL_COLOR;
    }

    /* (non-Javadoc)
     * @see paxion.Selectable#setSelected(boolean)
     */
    public void setSelected(boolean selected) {
        this.selected = selected;

        if (selected)
            currentColor = getSelectedColor();
        else
            currentColor = getNormalColor();

        repaint();
    }

    /* (non-Javadoc)
     * @see paxion.Selectable#isSelected()
     */
    public boolean isSelected() {
        return selected;
    }

    /**
     * Repaints this component.
     */
    public void repaint() {
```

```
        if (drawingPanel != null)
            drawingPanel.repaintScaled(bounds.x, bounds.y, bounds.width, bounds.
                height);
    }

    /**
     * Returns the x coordinate of this component.
     *
     * @return the x coordinate of this component origin (top-left corner)
     */
    public int getX() {
        return bounds.x;
    }

    /**
     * Returns the y coordinate of this component.
     *
     * @return the y coordinate of this component origin (top-left corner)
     */
    public int getY() {
        return bounds.y;
    }

    /**
     * Returns the width of this component.
     *
     * @return the width of this component
     */
    public int getWidth() {
        return bounds.width;
    }

    /**
     * Returns the height of this component.
     *
     * @return the height of this component
     */
    public int getHeight() {
        return bounds.height;
    }

    /**
     * Moves this component to the specified position.
     *
     * @param x the new x coordinate for the origin (top-left corner) of this
         component
     * @param y the new y coordinate for the origin (top-left corner) of this
         component
     */
    public void setPosition(int x, int y) {
        setBounds(x, y, bounds.width, bounds.height);
    }

    /**
     * Moves this component to the specified position.
     *
     * @param p the Point defining the new position of this component origin (top-
         left corner)
     */
    public void setPosition(Point p) {
        setPosition(p.x, p.y);
    }

    /**
     * Returns the position of this component.
     *
     * @return a new Point containing the position of this component origin (top-
         left corner)
```

```
     */
    public Point getPosition() {
        return new Point(bounds.x, bounds.y);
    }

    /**
     * Translates this component by the specified delta's.
     *
     * @param dx the difference between the new x coordinate and the current one
     * @param dy the difference between the new y coordinate and the current one
     */
    public void translate(int dx, int dy) {
        setPosition(bounds.x+dx, bounds.y+dy);
    }

    /**
     * Resizes this component to the specified size.
     *
     * @param w the new width for this component
     * @param h the new height for this component
     */
    public void setSize(int w, int h) {
        setBounds(bounds.x, bounds.y, w, h);
    }

    /**
     * Sets this component bounds to the specified x, y, w and h. This method can
        be
     * used to resize and move the component at the same time.
     *
     * @param x the new x coordinate for the origin (top-left corner) of this
        component
     * @param y the new y coordinate for the origin (top-left corner) of this
        component
     * @param w the new width for this component
     * @param h the new height for this component
     */
    public void setBounds(int x, int y, int w, int h) {
        repaint();
        bounds.x = x;
        bounds.y = y;
        bounds.width = w;
        bounds.height = h;
        repaint();
    }

    /**
     * Sets this component bounds to match the specified Rectangle. This method can
        be
     * used to resize and move the component at the same time.
     *
     * @param r the Rectangle specifying the new bounds
     */
    public void setBounds(Rectangle r) {
        setBounds(r.x, r.y, r.width, r.height);
    }

    /**
     * Returns the bounds of this component in a new Rectangle object.
     *
     * @return a new Rectangle containing the bounds of this component
     * @see #getBounds(Rectangle)
     */
    public Rectangle getBounds() {
        return new Rectangle(bounds);
    }

    /**
```

```
     * Returns the bounds of this component in the specified Rectangle.
     *
     * @param r the rectangle object where to store the bounds
     * @return the rectangle passed in parameter updated with the bounds of this
       component
     */
    public Rectangle getBounds(Rectangle r) {
        r.setBounds(bounds);
        return r;
    }

    /**
     * Returns wether the point at position (x,y) is inside this component or not.
     * Components with non rectangular bounds should override this method.
     *
     * @param x the x coordinate of the position to test
     * @param y the y coordinate of the position to test
     * @return true if the component contains the point (x,y); false otherwise.
     */
    public boolean contains(int x, int y) {
        return bounds.contains(x, y);
    }

    /**
     * Sets the DrawingPanel for this component.
     *
     * @param dp the new DrawingPanel
     */
    public void setDrawingPanel(DrawingPanel dp) {
        drawingPanel = dp;
    }

    /**
     * Draws this component using the specified graphics context.
     * Note that it is the component's responsibility to draw itself within its
       bounds.
     *
     * @param g2 the graphics context to use to draw the component
     */
    public abstract void draw(Graphics2D g2);

    /* (non-Javadoc)
     * @see paxion.Deletable#delete()
     */
    public void delete() {
        if (drawingPanel != null)
            drawingPanel.removeComponent(this);
    }

    /* (non-Javadoc)
     * @see paxion.HasContextMenu#showContextMenu(int, int)
     */
    public void showContextMenu(int x, int y) {
        contextMenu.show(drawingPanel, x, y);
    }

    /* (non-Javadoc)
     * @see paxion.Editable#edit()
     */
    public abstract void edit();

    /*
     * This is a convenience method for subclasses.
     */
    protected Frame getFrame() {
        Container c = drawingPanel.getTopLevelAncestor();

        if (!(c instanceof Frame)) {
```

```
            System.err.println("Warning: Couldn't find root window.");
            c = null;
        }

        return (Frame)c;
    }

    /*
     * Subclasses should override this method if they don't want edit and delete
     * items on their context menu.
     */
    protected AbstractAction[] getContextMenuActions() {
        return new AbstractAction[] {editAction, deleteAction};
    }

    /**
     * Creates the context/popup menu for this component.
     *
     * @param actions the actions that will form this context menu.
     * @return the new context menu
     */
    private JPopupMenu createContextMenu(AbstractAction[] actions) {
        JPopupMenu menu = new JPopupMenu();

        for (int i = 0; i < actions.length; i++)
            menu.add(actions[i]);

        return menu;
    }
}
```

C.2 AcceptStatesSet

Listing C.2: Source code for AcceptStatesSet

```
package paxion;

import java.io.*;
import java.util.*;

public class AcceptStatesSet {
    private static int nextSetId = 0;

    private Collection states = new HashSet();
    private int id;

    public AcceptStatesSet() {
        this(nextSetId);
    }

    public AcceptStatesSet(int id) {
        this.id = id;
        if (id >= nextSetId)
            nextSetId = id + 1;
    }

    public int getId() {
        return id;
    }

    public void addState(State s) {
        states.add(s);
    }
```

```
    public void removeState(State s) {
        states.remove(s);
    }

    public Collection getStates() {
        return states;
    }

    public void setCollection(Collection c) {
        states = c;
    }

    public void writeXML(PrintWriter writer) {
        Iterator i;

        writer.print("<acceptset id=\"" + id + "\">");

        i = states.iterator();
        while (i.hasNext()) {
            writer.print(Integer.toString(((State)i.next()).getId()));
            if (i.hasNext())
                writer.print(":");
        }

        writer.println("</acceptset>");
    }

    public void writeDivine(PrintWriter writer) {
        Iterator i;

        writer.print("accept ");
        i = states.iterator();
        while (i.hasNext()) {
            ((State)i.next()).writeDivine(writer);
            if (i.hasNext())
                writer.print(", ");
        }
        writer.println(";");
    }

    public String toString() {
        return "set " + id;
    }
}
```

C.3 AcceptStatesSetListPanel

Listing C.3: Source code for AcceptStatesSetListPanel

```
package paxion;

import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;

public class AcceptStatesSetListPanel extends EditableListPanel {
    protected ProcessEditor processEditor;

    protected JButton btSave = createSaveButton();
    protected JButton btCancel = createCancelButton();
    protected JPanel alternateButtonPanel = createAlternateButtonPanel();

    protected Collection temporarySet;
```

```
private int oldDrawingPanelMode;
private Collection oldSelectionCollection;

public AcceptStatesSetListPanel(String title, VectorListModel v) {
    super(title, v, AcceptStatesSet.class);

    alternateButtonPanel.setVisible(false);

    add(alternateButtonPanel);
}

protected JPanel createAlternateButtonPanel() {
    JPanel panel;

    panel = new JPanel();
    panel.setLayout(new BoxLayout(panel, BoxLayout.LINE_AXIS));

    // Save button
    panel.add(btSave);

    // Empty space
    panel.add(Box.createRigidArea(new Dimension(5, 0)));

    // Cancel button
    panel.add(btCancel);

    return panel;
}

protected JButton createNewButton() {
    JButton button;

    button = new JButton("new");
    button.setAlignmentX(Component.CENTER_ALIGNMENT);
    button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            AcceptStatesSet newSet = new AcceptStatesSet();
            data.add(newSet);
        }
    });

    return button;
}

protected JButton createEditButton() {
    JButton button;

    button = new JButton("edit");
    button.setAlignmentX(Component.CENTER_ALIGNMENT);
    button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                AcceptStatesSet selectedValue = (AcceptStatesSet)list.
                    getSelectedValue();
                if (selectedValue != null) {
                    temporarySet = new HashSet(selectedValue.getStates());
                    oldSelectionCollection = processEditor.getDrawingPanel().
                        getSelectionCollection();
                    oldDrawingPanelMode = processEditor.getDrawingPanel().
                        getMode();
                    processEditor.getDrawingPanel().setMode(DrawingPanel.
                        ACCEPTSTATE_MODE);
                    processEditor.getDrawingPanel().setSelectionCollection(
                        temporarySet);
                    buttonPanel.setVisible(false);
                    alternateButtonPanel.setVisible(true);
                    list.setEnabled(false);
                }
```

```
                }
            });

        return button;
    }

    protected JButton createSaveButton() {
        JButton button;

        button = new JButton("save");
        button.setAlignmentX(Component.CENTER_ALIGNMENT);
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                AcceptStatesSet selectedValue = (AcceptStatesSet)list.
                    getSelectedValue();

                // this test is theorically useless but well... you never know...
                if (selectedValue != null) {
                    selectedValue.setCollection(temporarySet);
                }

                stopEditing();
            }
        });

        return button;
    }

    protected JButton createCancelButton() {
        JButton button;

        button = new JButton("cancel");
        button.setAlignmentX(Component.CENTER_ALIGNMENT);
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                stopEditing();
            }
        });

        return button;
    }

    public void stopEditing() {
        // Changes the collection used for the selection (restores the normal
            collection)
        processEditor.getDrawingPanel().setSelectionCollection(
            oldSelectionCollection);
        processEditor.getDrawingPanel().setMode(oldDrawingPanelMode);
        list.setEnabled(true);
        alternateButtonPanel.setVisible(false);
        buttonPanel.setVisible(true);
    }

    public void addNotify() {
        super.addNotify();

        for (Container c = getParent(); c != null; c = c.getParent())
            if (c instanceof ProcessEditor)
                processEditor = (ProcessEditor)c;

        if (processEditor == null) {
            System.err.println(
                "AcceptStatesSetListPanel Components should only be used within
                    ProcessEditors!");
            System.exit(1);
        }
    }
}
```

C.4 AnchorableLabel

Listing C.4: Source code for AnchorableLabel

```
package paxion;

import java.io.*;
import java.awt.*;

public class AnchorableLabel extends CenteredComponent {
    // this is supposed to be a reference to a point updated by the parent object
    // of this label
    protected Point anchorPoint;
    protected Point deltaToAnchor;
    protected String text;
    protected AbstractComponent parent;

    private FontMetrics metrics = null;

    private int width;
    private int height;
    private int ascent;

    public AnchorableLabel(String text) {
        this(text, null, null);
    }

    public AnchorableLabel(String text, Point deltaToAnchor) {
        this(text, deltaToAnchor, null);
    }

    public AnchorableLabel(String text, Point delta, Point anchor) {
        this.text = text;

        if (delta != null)
            deltaToAnchor = delta;
        else
            deltaToAnchor = new Point();

        if (anchor != null)
            anchorPoint = anchor;
        else
            anchorPoint = new Point();

        setAnchorPoint(anchorPoint);
        setDeltaToAnchorPoint(deltaToAnchor);
    }

    public void setParent(AbstractComponent c) {
        parent = c;
    }

    public AbstractComponent getParent() {
        return parent;
    }

    public void setText(String text) {
        if (text.equals(""))
            delete();
        else {
            this.text = text;
```

```
            if (metrics != null) {
                /* stringWidth is inaccurate. that's why we need the +1. Ideally we
                    should
                 * rather use getStringBounds(String str, Graphics context) but
                    this is
                 * troublesome and stringWidth(String str) works well in most cases
                    .
                 */
                width = metrics.stringWidth(text)+1;
                setSize(width, height);
                repaint();
            }
        }
    }

    public String getText() {
        return text;
    }

    public void setAnchorPoint(Point anchorPoint) {
        this.anchorPoint = anchorPoint;
        update();
    }

    public void setDeltaToAnchor(int x, int y) {
        deltaToAnchor.x = x;
        deltaToAnchor.y = y;
        setCenter(x + anchorPoint.x, y + anchorPoint.y);
    }

    public void setDeltaToAnchorPoint(Point p) {
        deltaToAnchor = p;
        setDeltaToAnchor(p.x, p.y);
    }

    public void update() {
        if (deltaToAnchor != null)
            setCenter(deltaToAnchor.x + anchorPoint.x, deltaToAnchor.y +
                anchorPoint.y);
        else
            setCenter(anchorPoint.x, anchorPoint.y);
    }

    public void setPosition(int x, int y) {
        deltaToAnchor.x = x + deltaToCenterX - anchorPoint.x;
        deltaToAnchor.y = y + deltaToCenterY - anchorPoint.y;
        super.setPosition(x, y);
    }

    public void setSize(int w, int h) {
        setDeltaToCenter(w / 2, h / 2);
        super.setSize(w, h);
    }

    public void updateMetrics(FontMetrics newMetrics) {
        metrics = newMetrics;
        ascent = metrics.getMaxAscent();
        height = ascent + metrics.getMaxDescent();
        width = metrics.stringWidth(text);
        setSize(width, height);
    }

    public void edit() {
        LabelEditor le = new LabelEditor(null, this);
    }

    public void draw(Graphics2D g2) {
        g2.setPaint(currentColor);
```

```
        g2.drawString(text, getX(), getY() + ascent);
    }

    public void setDrawingPanel(DrawingPanel dp) {
        super.setDrawingPanel(dp);

        Graphics g = dp.getGraphics();
        if (g == null) {
            System.err.println("Warning: Couldn't get graphic context for label!");
            return;
        } else {
            updateMetrics(g.getFontMetrics());
            update();
        }
    }

    public void writeXML(PrintWriter writer) {
        writer.println("text=\"" + Utils.convertInvalidXMLChars(text) + "\">");

        if (deltaToAnchor != null)
            writer.println("<point x=\"" + deltaToAnchor.x + "\" y=\"" +
                deltaToAnchor.y + "\"/>");
    }

    public void writeDivine(PrintWriter writer) {
        writer.print(text);
    }
}
```

C.5 Arrow

Listing C.5: Source code for Arrow

```
package paxion;

import java.awt.*;
import java.util.*;
import java.awt.geom.*;
import java.awt.event.*;
import javax.swing.*;

public class Arrow {
    private static BasicStroke stroke = new BasicStroke(1.0f);
    private Color color = Color.black;

    private static final float size1 = 10.0f;
    private static final float size2 = 5.0f;

    private GeneralPath endPath = new GeneralPath();
    private GeneralPath mainPath = new GeneralPath();

    private Shape endShape;
    private Shape mainShape;

    private AffineTransform at = new AffineTransform();
    private Rectangle bb;

    private LinkedList points = new LinkedList();

    public Arrow() {
        endPath.moveTo(-size1, +size2);
        endPath.lineTo(0, 0);
        endPath.lineTo(-size1, -size2);
    }
```

```
public Arrow(Point o, Point d) {
    this();

    points.add(o);
    points.add(d);

    update();
}

public void addPoint(Point p) {
    points.add(p);
}

public void insertPointBeforeLast(Point p) {
    points.add(points.size() - 1, p);
}

public void removePoint(Point p) {
    points.remove(p);
}

public int getNumberPoints() {
    return points.size();
}

public LinkedList getPoints() {
    return points;
}

public Point getPoint(int index) {
    return (Point)points.get(index);
}

public void setDest(int x, int y) {
    ((Point)points.getLast()).setLocation(x, y);
}

public Point getOrigin() {
    return (Point)points.getFirst();
}

public Point getDest() {
    return (Point)points.getLast();
}

public void update() {
    int nbrPoints = points.size();

    if (nbrPoints < 2)
        return;

    // calculate angle of last segment

    Point p1 = (Point)points.get(nbrPoints-2);
    Point p2 = (Point)points.get(nbrPoints-1);

    double theta = Math.atan2(p2.getY() - p1.getY(), p2.getX() - p1.getX());

    at.setToTranslation(p2.getX(), p2.getY());
    at.rotate(theta);

    endShape = at.createTransformedShape(endPath);

    // create main path
    mainPath.reset();

    Iterator i = points.iterator();
```

```
        Point p = (Point)i.next();
        mainPath.moveTo((float)p.getX(), (float)p.getY());

        if (nbrPoints == 2) {

            // if there are only 2 points, we can draw a straight line
            p = (Point)i.next();
            mainPath.lineTo((float)p.getX(), (float)p.getY());

        } else {

            // if there are more ...
            int prevX = (int)p.getX();
            int prevY = (int)p.getY();

            p = (Point)i.next();
            int curX = (int)p.getX();
            int curY = (int)p.getY();

            int midX = (curX + prevX) / 2;
            int midY = (curY + prevY) / 2;

            // ... we start by drawing the half of the first segment (straight line
                )
            mainPath.lineTo(midX, midY);

            // ... then we draw curves for the next segments ...
            while (i.hasNext()) {
                prevX = curX;
                prevY = curY;

                p = (Point)i.next();
                curX = (int)p.getX();
                curY = (int)p.getY();

                midX = (curX + prevX) / 2;
                midY = (curY + prevY) / 2;

                mainPath.quadTo(prevX, prevY, midX, midY);
            }

            // ... until we reach the last one for which we draw the second half
                with
            // a straight line.
            mainPath.lineTo(curX, curY);
        }

        mainShape = mainPath;
    }

    public Rectangle getBounds() {
        bb = mainShape.getBounds();
        bb.add(endShape.getBounds());
        bb.grow(1, 1);
        return bb;
    }

    public boolean contains(int x, int y) {
        int nbrPoints = points.size();

        if (nbrPoints < 2)
            return false;

        // For each segment, we check if the point is near it.
        PathIterator p = mainPath.getPathIterator(null);
        FlatteningPathIterator f = new FlatteningPathIterator(p, 0.1); // flatness

        if (f.isDone())
```

```
            return false;

        float[] pts = new float[6];
        float prevX, prevY;

        f.currentSegment(pts);
        f.next();

        while ( !f.isDone() ) {
            prevX = pts[0];
            prevY = pts[1];

            if ( f.currentSegment(pts) == PathIterator.SEG_LINETO )
                if (Line2D.ptSegDist(prevX, prevY, pts[0], pts[1], x, y) < 3)
                    return true;

            f.next();
        }

        return false;
    }

    public void draw(Graphics2D g2) {
        g2.setStroke(stroke);
        g2.setPaint(color);

        g2.draw(mainShape);
        g2.draw(endShape);
    }

    public void setColor(Color color) {
        this.color = color;
    }

    public String toString() {
        return "Arrow[]";
    }

    //-----------------------------------------------------------------------
    // testing code

    public static void main(String[] args) {
        JFrame frame;
        ContentPane contentPane;

        frame = new JFrame("Arrow test");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        contentPane = new ContentPane();
        contentPane.setOpaque(true);
        contentPane.setPreferredSize(new Dimension(500, 500));
        contentPane.setSize(new Dimension(500, 500));

        frame.setContentPane(contentPane);
        frame.pack();
        frame.setVisible(true);
    }

    private static class ContentPane extends JPanel implements MouseListener {
        private Arrow arrow = new Arrow();

        public ContentPane() {
            super(null);
            addMouseListener(this);

            Random generator = new Random();

            for (int i = 0; i<5; i++) {
```

```
                Point p = new Point(50 + generator.nextInt(400), 50 + generator.
                    nextInt(400));
                arrow.addPoint(p);
            }
            arrow.update();
        }

        public void paintComponent(Graphics g) {
            super.paintComponent(g);
            arrow.draw((Graphics2D)g);
        }

        public void mousePressed(MouseEvent e) {
            arrow.setDest(e.getX(), e.getY());
            arrow.update();
            repaint();
        }
        public void mouseReleased(MouseEvent e) {}
        public void mouseClicked(MouseEvent e) {}
        public void mouseEntered(MouseEvent e) {}
        public void mouseExited(MouseEvent e) {}
    };
}
```

C.6 CenteredComponent

Listing C.6: Source code for CenteredComponent

```
package paxion;

import java.awt.*;

public abstract class CenteredComponent extends AbstractComponent {
    // position of the "center" of the component (in world coordinate)
    protected Point center;

    // position of the "center" of the component within the component
    protected int deltaToCenterX = 0;
    protected int deltaToCenterY = 0;

    public CenteredComponent() {
        this(new Point());
    }

    public CenteredComponent(Point p) {
        setCenterPoint(p);
    }

    /* (non-Javadoc)
     * @see paxion.AbstractComponent#setBounds(int, int, int, int)
     */
    public void setBounds(int x, int y, int w, int h) {
        center.x = x + deltaToCenterX;
        center.y = y + deltaToCenterY;
        super.setBounds(x, y, w, h);
    }

    /**
     * Move the component by specifying the new coordinates for its center point
     *
     * @param x
     * @param y
     */
    public void setCenter(int x, int y) {
```

```
        center.x = x;
        center.y = y;
        super.setPosition(x-deltaToCenterX, y-deltaToCenterY);
    }

    /**
     * @return a copy of the center point
     */
    public Point getCenter() {
        return new Point(center);
    }

    /**
     * Sets a new point as the center point (use with care!)
     *
     * @param p
     */
    public void setCenterPoint(Point p) {
        center = p;
        setCenter(p.x, p.y);
    }

    /**
     * @return a reference to the center point (use with care!)
     */
    public Point getCenterPoint() {
        return center;
    }

    public void setDeltaToCenter(int x, int y) {
        if ((deltaToCenterX != 0) || (deltaToCenterY != 0))
            translate(deltaToCenterX, deltaToCenterY);

        deltaToCenterX = x;
        deltaToCenterY = y;

        if ((x != 0) || (y != 0))
            translate(-x, -y);
    }

    public void setDeltaToCenter(Point p) {
        setDeltaToCenter(p.x, p.y);
    }

    public Point getDeltaToCenter() {
        return new Point(deltaToCenterX, deltaToCenterY);
    }
}
```

C.7 ChannelEditor

Listing C.7: Source code for ChannelEditor

```
package paxion;

import java.awt.*;
import javax.swing.*;

public class ChannelEditor extends ObjectEditor {
    private JTextField tfName;

    public ChannelEditor(Frame frame, Channel channel) {
        super(frame, "Edit Channel", channel);
    }
```

```
    protected JPanel createObjectPropertiesPanel() {
        JPanel panel;
        JLabel label;

        panel = new JPanel();
        panel.setLayout(new BoxLayout(panel, BoxLayout.LINE_AXIS));
        panel.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));

        label = new JLabel("name:");
        tfName = new JTextField(((Channel)object).getName());
        tfName.setMaximumSize(new Dimension(Short.MAX_VALUE, Short.MAX_VALUE));

        panel.add(label);
        panel.add(Box.createRigidArea(new Dimension(10, 0)));
        panel.add(tfName);

        return panel;
    }

    protected void updateObject() {
        // this prevents setting a channel name to ""
        if (!tfName.getText().equals(""))
            ((Channel)object).setName(tfName.getText());
    }
}
```

C.8 Channel

Listing C.8: Source code for Channel

```
package paxion;

import java.io.PrintWriter;

public class Channel implements Editable {
    private String name;

    public Channel() {
        this("");
    }

    public Channel(String name) {
        setName(name);
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void edit() {
        ChannelEditor ce = new ChannelEditor(null, this);
    }

    public void writeXML(PrintWriter writer) {
        writer.println("<channel name=\"" + name + "\"/>");
    }

    public void writeDivine(PrintWriter writer) {
        writer.print(name);
```

```
    }

    public String toString() {
        return name;
    }
}
```

C.9 CommonAction

Listing C.9: Source code for CommonAction

```
package paxion;

import javax.swing.*;

public abstract class CommonAction extends AbstractAction {
    public ImageIcon largeIcon;
    public CommonAction(String text, String iconPath, String desc, Integer mnemonic
        , KeyStroke accel) {
        putValue(NAME, text);
        putValue(SHORT_DESCRIPTION, desc);
        putValue(MNEMONIC_KEY, mnemonic);
        putValue(ACCELERATOR_KEY, accel);
        putValue(SMALL_ICON, Utils.loadIcon(iconPath, true));
        largeIcon = Utils.loadIcon(iconPath, false);
    }
}
```

C.10 Deletable

Listing C.10: Source code for Deletable

```
package paxion;

public interface Deletable {
    /**
     * Removes the component from the {@link DrawingPanel} and notify other
     * objects which point to this component that it doesn't exist anymore.
     */
    void delete();
}
```

C.11 DivineFileFilter

Listing C.11: Source code for DivineFileFilter

```
package paxion;

import java.io.File;
import javax.swing.filechooser.*;

public class DivineFileFilter extends FileFilter {

    // Accept all .dve files and directories
    public boolean accept(File f) {
```

```
        if (f.isDirectory())
            return true;

        String extension = Utils.getExtension(f);
        if (extension != null) {
            if (extension.equals("dve"))
                return true;
            else
                return false;
        }

        return false;
    }

    // The description of this filter
    public String getDescription() {
        return "Divine Files (*.dve)";
    }
}
```

C.12 DrawingPanel

Listing C.12: Source code for DrawingPanel

```
package paxion;

import java.awt.*;
import java.awt.geom.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;

public class DrawingPanel extends JPanel implements MouseListener,
    MouseMotionListener {
    public static final int SELECT_MODE = 0;
    public static final int STATE_MODE = 1;
    public static final int TRANSITION_MODE = 2;
    public static final int ACCEPTSTATE_MODE = 3;

    public static final float DEFAULT_SCALING = 1.0f;
    public static final float ZOOM_FACTOR = 2.0f;

    private static final int GRID_STEP = 50;
    private static final Color GRID_COLOR = Color.LIGHT_GRAY;
    private static final float DASH[] = { 2.0f };
    private static final BasicStroke GRID_STROKE =
        new BasicStroke(1.0f, BasicStroke.CAP_BUTT, BasicStroke.JOIN_MITER,
            10.0f, DASH, 0.0f);

    private static final BasicStroke SELECTION_RECTANGLE_STROKE = new BasicStroke
        (1.0f);

    public CommonAction setSelectModeAction, setStateModeAction,
        setTransitionModeAction,
        zoomInAction, zoomOutAction,
        toggleGridAction;

    private JPopupMenu contextMenu;

    private Rectangle drawingArea = new Rectangle();
    private boolean showGrid = true;

    private int currentMode = STATE_MODE;
```

```
    private State fromState = null;
    private Arrow arrow = null;

    private float scaling = DEFAULT_SCALING;

    private LinkedList componentList = new LinkedList();

    // selection
    private Point selectionPoint = null;
    private Rectangle selectionRectangle = null;
    private Collection selectionCollection = new HashSet();

    // for mouse
    private AbstractComponent lastComponentClicked = null;
    private AbstractComponent activeComponent = null;
    private Point lastPointPressed = new Point();
    private boolean dragging = false;

    //
    private Process process;

    public DrawingPanel(Process p) {
        super();

        process = p;

        setBackground(Color.white);
        setLayout(null);
        setOpaque(true);

        addMouseListener(this);
        addMouseMotionListener(this);

        getInputMap(WHEN_FOCUSED).put(KeyStroke.getKeyStroke(KeyEvent.VK_DELETE, 0)
            , "delete");

        Action myAction = new AbstractAction("delete") {
                public void actionPerformed(ActionEvent e) {
                    if (getMode() != ACCEPTSTATE_MODE)
                        deleteSelected();
                }
            };

        getActionMap().put(myAction.getValue(Action.NAME), myAction);

        // Create the actions
        createActions();

        // Create the context menu
        contextMenu = createContextMenu();
    }

    public Process getProcess() {
        return process;
    }

    /*
     * @see ProjectEditor#addProcess(Process process) for an explanation of why
         this method
     * can't be merged nor called in the constructor.
     */
    public void update() {
        Iterator i;

        /* Remove all components from the panel. This is useless with the current
         * use of method (just after the DrawingPanel was created) but could prove
         * usefull in the future.
         */
```

```
        componentList.clear();

        // add states to the back of the panel (and the labels in front of them)
        i = process.getStates().iterator();
        while (i.hasNext())
            addState((State)i.next());

        // add the transitions on top of the rest
        i = process.getTransitions().iterator();
        while (i.hasNext())
            addTransition((Transition)i.next());
    }

    /**
     * Returns the top component containing the point at coordinates (x,y). The top
         component
     * means that if several components include this point, the closest to the top
         will be
     * returned. As a consequence of this, the component returned is always visible
         at
     * this point.
     *
     * @param x the x coordinate of the point to test
     * @param y the y coordinate of the point to test
     * @return the component containing the point at the specified coordinates
     */
    public AbstractComponent getComponent(int x, int y) {

        /* We iterate on the list of components in reverse order which correspond
            to
         * from "front" to "back"
         */
        ListIterator i = componentList.listIterator(componentList.size());
        while (i.hasPrevious()) {
            AbstractComponent c = (AbstractComponent)i.previous();
            if (c.contains(x, y))
                return c;
        }

        return null;
    }

    public void translate(int dx, int dy) {
        Iterator i;

        /* Translating States and TransitionPoints is enough because all the other
            components
         * "depend" (directly or indirectly) on states (and they are updated
             automatically when
         * a state is moved)
         */
        i = componentList.iterator();
        while (i.hasNext()) {
            AbstractComponent c = (AbstractComponent)i.next();
            Class cl = c.getClass();
            if ((cl == State.class) || (cl == TransitionPoint.class))
                c.translate(dx, dy);
        }
    }

    public void toggleGrid() {
        showGrid = !showGrid;
        repaint();
    }

    //---------------

    public void setMode(int mode) {
```

```
        currentMode = mode;

        // Deselect all selected objects
        clearSelection();

        if (fromState != null) {
            fromState = null;
            arrow = null;
            repaint();
        }
    }

    public int getMode() {
        return currentMode;
    }

    //---------------------

    public void zoomIn() {
        setScaling(scaling * ZOOM_FACTOR);
    }

    public void zoomOut() {
        setScaling(scaling / ZOOM_FACTOR);
    }

    //--

    public void setScaling(float s) {
        float oldscaling = scaling;

        JViewport viewPort = (JViewport)getParent();
        Rectangle vr = viewPort.getViewRect();

        scaling = s;

        drawingArea.width *= scaling/oldscaling;
        drawingArea.height *= scaling/oldscaling;

        updateSizeAndPreferredSize();

        /* The following code tries to center view (adjust the viewPort's
            viewPosition).
         * It doesn't always work as expected but the problem is not here but
            rather
         * due to the way the drawingArea is managed (and especially expanded).
         */
        int diffwidth =
            (int) (vr.width / oldscaling) - (int) (vr.width / scaling);
        int diffheight =
            (int) (vr.height / oldscaling) - (int) (vr.height / scaling);

        Point p =
            new Point(
                (int) (vr.x * scaling / oldscaling)
                    + (int) (diffwidth * scaling / 2),
                (int) (vr.y * scaling / oldscaling)
                    + (int) (diffheight * scaling / 2));

        viewPort.setViewPosition(p);
    }

    public float getScaling() {
        return scaling;
    }

    //---------------------
```

```
public void setTransitionExtremity(State s) {
    if (fromState == null) {
        Point p1 = new Point(s.getCenter());
        Point p2 = new Point(s.getCenter());
        arrow = new Arrow(p1, p2);
        fromState = s;
    } else {
        repaintScaled(arrow.getBounds());
        addTransition(new Transition(fromState, s, arrow));
        fromState = null;
        arrow = null;
    }
}

public void addArrowPoint(int x, int y) {
    if (arrow == null)
        return;
    arrow.addPoint(new Point(x,y));
}

public void setArrowDest(int x, int y) {
    if (arrow == null)
        return;

    Rectangle r = arrow.getBounds();
    arrow.setDest(x, y);
    arrow.update();
    repaintScaled(r.union(arrow.getBounds()));
}

//----------------

public void paintComponent(Graphics g) {
    Graphics2D g2 = (Graphics2D) g;

    AffineTransform saveAT = g2.getTransform();

    super.paintComponent(g);

    if (showGrid) {
        g2.setStroke(GRID_STROKE);
        g2.setPaint(GRID_COLOR);

        int step = (int)(GRID_STEP * scaling);

        for (int x = 0; x <= getWidth(); x += step)
            g2.drawLine(x, 0, x, getHeight());

        for (int y = 0; y <= getHeight(); y += step)
            g2.drawLine(0, y, getWidth(), y);
    }

    // turn on antialiasing
    g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.
        VALUE_ANTIALIAS_ON);

    if (scaling != 1.0)
        g2.scale(scaling, scaling);

    Iterator i = componentList.iterator();
    while (i.hasNext())
        ((AbstractComponent)i.next()).draw(g2);

    if (arrow != null)
        arrow.draw(g2);

    // Restore original transform
    g2.setTransform(saveAT);
```

```
        if (selectionRectangle != null) {
            Rectangle r = new Rectangle(selectionRectangle);
            r.x *= scaling;
            r.y *= scaling;
            r.width *= scaling;
            r.height *= scaling;
            g2.setStroke(SELECTION_RECTANGLE_STROKE);
            g2.setPaint(Color.BLUE);
            g2.draw(r);
        }
    }

    //-----------------

    public void printComponent(Graphics g) {
        Graphics2D g2 = (Graphics2D) g;

        AffineTransform saveAT = g2.getTransform();

        if (scaling != 1.0)
            g2.scale(scaling, scaling);

        Iterator i = componentList.iterator();
        while (i.hasNext()) {
            AbstractComponent c = (AbstractComponent)i.next();
            if (!(c instanceof TransitionPoint))
                c.draw(g2);
        }

        // Restore original transform
        g2.setTransform(saveAT);
    }

    //-----------------

    public void addState(State s) {
        // add the new state to the process
        process.addState(s);

        // add the new state at the back of the panel
        addComponent(s, true);
    }

    public void addTransition(Transition t) {
        // add the new transition to the process
        process.addTransition(t);

        // add the new transition as the top most component of the panel
        addComponent(t, false);
    }

    public void addLabel(AnchorableLabel l) {
        addComponent(l, false);
    }

    //-----------------

    public void clearSelection() {
        Iterator i = selectionCollection.iterator();
        while (i.hasNext())
            ((Selectable)i.next()).setSelected(false);
        selectionCollection.clear();
    }

    //-----------------

    public void addToSelection(Selectable s) {
```

```
        if (s == null)
            return;

        s.setSelected(true);
        selectionCollection.add(s);
    }

    public void removeFromSelection(Selectable s) {
        if (s == null)
            return;

        s.setSelected(false);
        selectionCollection.remove(s);
    }

    public void setSelection(Selectable s) {
        clearSelection();
        addToSelection(s);
    }

    public void toggleSelection(Selectable c) {
        if (c == null)
            return;

        if (c.isSelected())
            removeFromSelection(c);
        else
            addToSelection(c);
    }

    //-----------------

    public void addToSelection(Rectangle r) {
        if (r == null)
            return;

        Rectangle bounds = new Rectangle();
        Iterator i = componentList.iterator();

        while (i.hasNext()) {
            AbstractComponent c = (AbstractComponent)i.next();
            c.getBounds(bounds);
            if (r.contains(bounds))
                addToSelection(c);
        }
    }

    public void setSelection(Rectangle r) {
        clearSelection();
        addToSelection(r);
    }

    public void toggleSelection(Rectangle r) {
        if (r == null)
            return;

        Rectangle bounds = new Rectangle();
        Iterator i = componentList.iterator();
        while (i.hasNext()) {
            AbstractComponent c = (AbstractComponent)i.next();
            c.getBounds(bounds);
            if (r.contains(bounds))
                toggleSelection(c);
        }
    }

    //-----------------
```

```
    /**
     * Replaces the Collection object used for selection. It doesn't clear the old
     * collection but it is the caller responsibility to keep a link to the
     * old Collection object if needed.
     *
     * @param c
     * @see #getSelectionCollection()
     */
    public void setSelectionCollection(Collection c) {
        if (selectionCollection == c)
            return;

        Iterator i;

        i = selectionCollection.iterator();
        while (i.hasNext())
            ((Selectable)i.next()).setSelected(false);

        selectionCollection = c;

        i = selectionCollection.iterator();
        while (i.hasNext())
            ((Selectable)i.next()).setSelected(true);
    }

    /**
     * Returns a reference to the Collection object used for the selection.
     * Use with care!
     *
     * @return a reference to the current Collection object used for the selection.
     */
    public Collection getSelectionCollection() {
        return selectionCollection;
    }

    //------------------

    public void deleteSelected() {
        /*
         * We iterate through the selectionCollection and delete each component
         * if it is not going to be deleted because of another component
         *
         * Components which can be deleted by other components are:
         *      - StateLabels: can be deleted if their state is deleted
         *      - Transitions labels (Guards, Synchros, Effects) and ArrowPoints:
         *          can be deleted if:
         *          * their transition is deleted
         *          * one of the two extremity states of their transition is deleted
         *      - Transitions: can be deleted if one of their two extremity states
         *    is deleted
         *
         * These components are thus skipped from being deleted if (one of) the
         *    component(s)
         * which can cause them to be deleted also selected.
         */

        Iterator i = selectionCollection.iterator();
        while (i.hasNext()) {
            AbstractComponent c = (AbstractComponent)i.next();
            AbstractComponent parent = null;
            Transition t = null;

            if (c instanceof AnchorableLabel)
                parent = ((AnchorableLabel)c).getParent();
            else if (c instanceof TransitionPoint)
                parent = ((TransitionPoint)c).getTransition();

            if (parent != null)
```

```
            if (parent.isSelected())
                continue;    // skip this component

        if (c instanceof Transition)
            t = (Transition)c;
        else if (parent instanceof Transition)
            t = (Transition)parent;

        if (t != null)
            if (t.getOrigin().isSelected() || t.getDestination().isSelected())
                continue;    // skip this component

        ((Deletable)c).delete();
    }

    selectionCollection.clear();
}

//-----------------

public void dragSelected(int dx, int dy) {
    /*
     * We iterate through the selectionCollection and for each component we
       translate it
     * if it is dragable AND if it is not going to be "moved" by another
       component.
     *
     * Components which can be "moved" by other components are all labels:
     *      StateLabels can be moved by their state
     *      Transitions labels (Guards, Synchros, Effects) can be moved by the
       two states
     *          at the extremities of their transition
     *
     * These labels are thus skipped from being translated if (one of) the
       component(s)
     * which can move them is also selected.
     *
     * Note: that this works independently from the iteration order (wether the
       "child"
     * comes before his "parent" or the opposite).
     */

    Iterator i = selectionCollection.iterator();
    while (i.hasNext()) {
        AbstractComponent c = (AbstractComponent)i.next();
        if (isClassDraggable(c.getClass())) {
            // Check if the component is a label and should be skipped or not
            if (c instanceof StateLabel) {
                if (((AnchorableLabel)c).getParent().isSelected())
                    continue;    // skip this label
            } else if ((c instanceof Guard) || (c instanceof Synchro) || (c
                instanceof Effect)) {
                Transition t = (Transition)((AnchorableLabel)c).getParent();
                if (t.getOrigin().isSelected() || t.getDestination().isSelected
                    ())
                    continue;    // skip this label
            }

            // Translate the component
            c.translate(dx, dy);

            /* Extend the drawing area if needed. Note that this could lead to
                a
             * translation of all components of the panel but
             * it's the way it should be.
             */
            extendDrawingArea(c);
        }
```

```
    }
}

//------------------
// Allow [component type] to be dragged in [mode]
// All AbstractComponents except Transitions in SELECT_MODE
// State and StateLabel in STATE_MODE
// ArrowPoints and "transition labels" in TRANSITION_MODE
// nothing in ACCEPTSTATE_MODE
public boolean isClassDraggable(Class c) {
    switch (getMode()) {
    case SELECT_MODE:
        return (AbstractComponent.class.isAssignableFrom(c) && !Transition.
            class.isAssignableFrom(c)) ;
    case STATE_MODE:
        return (State.class.isAssignableFrom(c) | StateLabel.class.
            isAssignableFrom(c));
    case TRANSITION_MODE:
        return (TransitionPoint.class.isAssignableFrom(c) | Guard.class.
            isAssignableFrom(c) | Effect.class.isAssignableFrom(c) | Synchro.
            class.isAssignableFrom(c));
    case ACCEPTSTATE_MODE:
        return false;
    }

    return false;
}

// Allow [component type] to be selected in [mode]
// Selectable in SELECT_MODE
// State and StateLabel in STATE_MODE
// TransitionPoint and "transitions labels" and Transition in TRANSITION_MODE
// State in ACCEPTSTATE_MODE
public boolean isClassSelectable(Class c) {
    switch (getMode()) {
    case SELECT_MODE:
        return Selectable.class.isAssignableFrom(c);
    case STATE_MODE:
        return (State.class.isAssignableFrom(c) | StateLabel.class.
            isAssignableFrom(c));
    case TRANSITION_MODE:
        return (TransitionPoint.class.isAssignableFrom(c) | Guard.class.
            isAssignableFrom(c) | Effect.class.isAssignableFrom(c) | Synchro.
            class.isAssignableFrom(c) | Transition.class.isAssignableFrom(c));
    case ACCEPTSTATE_MODE:
        return State.class.isAssignableFrom(c);
    }

    return false;
}

//------------------

public void mousePressed(MouseEvent e) {
    requestFocusInWindow();

    int x = e.getX();
    int y = e.getY();

    dragging = false;

    AbstractComponent c = getComponent(x, y);

    if (e.isPopupTrigger()) {
        if (c == null)
            contextMenu.show(this, (int)(x*scaling), (int)(y*scaling));
        else if (c instanceof HasContextMenu)
```

```
                ((HasContextMenu)c).showContextMenu((int)(x*scaling), (int)(y*
                    scaling));

            return;
        }

        if (c != null)
            if (isClassSelectable(c.getClass())) {
                lastPointPressed.x = x - c.getX();
                lastPointPressed.y = y - c.getY();

                if ((e.getModifiersEx() & InputEvent.CTRL_DOWN_MASK) == InputEvent.
                    CTRL_DOWN_MASK)
                    toggleSelection(c);
                else if (!c.isSelected())
                    setSelection(c);

                if (c.isSelected())
                    activeComponent = c;

                return;
            }

        switch (currentMode) {
        case SELECT_MODE:
            if (c == null) {
                selectionPoint = e.getPoint();
                selectionRectangle = new Rectangle(selectionPoint);
            }
            break;

        case STATE_MODE:
            addState(new State(x, y));
            break;

        case TRANSITION_MODE:
            if (c instanceof State) {
                setTransitionExtremity((State)c);
            } else {
                if (arrow != null)
                    addArrowPoint(x, y);
            }
            break;

        case ACCEPTSTATE_MODE:
            break;
        }
    }

    public void mouseReleased(MouseEvent e) {
        boolean wasDragging = dragging;
        dragging = false;

        int x = e.getX();
        int y = e.getY();

        AbstractComponent c = getComponent(x, y);
        if (e.isPopupTrigger()) {
            if (c == null)
                contextMenu.show(this, (int)(x*scaling), (int)(y*scaling));
            else if (c instanceof HasContextMenu)
                ((HasContextMenu)c).showContextMenu((int)(x*scaling), (int)(y*
                    scaling));

            return;
        }

        activeComponent = null;
```

```
        if ((c == null) && (selectionRectangle == null)) {
            // deselect all selected objects
            clearSelection();
        }

        switch (currentMode) {
        case SELECT_MODE:
            if (selectionRectangle != null) {
                if ((e.getModifiersEx() & InputEvent.CTRL_DOWN_MASK) == InputEvent.
                    CTRL_DOWN_MASK)
                    toggleSelection(selectionRectangle);
                else
                    setSelection(selectionRectangle);
                repaintScaled(selectionRectangle.x, selectionRectangle.y,
                    selectionRectangle.width+1, selectionRectangle.height+1);
                selectionRectangle = null;
                selectionPoint = null;
            }
            break;

        case STATE_MODE:
            break;

        case TRANSITION_MODE:
            if (wasDragging) {
                if (c instanceof State)
                    setTransitionExtremity((State)c);
                else
                    addArrowPoint(e.getX(), e.getY());
            }
            break;

        case ACCEPTSTATE_MODE:
            break;
        }
    }

    public void mouseClicked(MouseEvent e) {
        AbstractComponent c = getComponent(e.getX(), e.getY());
        if ((c instanceof Editable) && (c == lastComponentClicked) && (e.
            getClickCount() == 2))
            ((Editable)c).edit();
        lastComponentClicked = c;
    }

    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}

    public void mouseMoved(MouseEvent e) {
        if (arrow != null)
            setArrowDest(e.getX(), e.getY());
    }

    public void mouseDragged(MouseEvent e) {
        dragging = true;
        if (activeComponent != null) {
            if (isClassDraggable(activeComponent.getClass())) {
                int dx = e.getX() - activeComponent.getX() - lastPointPressed.x;
                int dy = e.getY() - activeComponent.getY() - lastPointPressed.y;
                dragSelected(dx, dy);
            }
        } else if (arrow != null) {
            setArrowDest(getX() + e.getX(), getY() + e.getY());
        } else if (selectionRectangle != null) {
            Rectangle oldRectangle = new Rectangle(selectionRectangle);

            selectionRectangle.setSize(0, 0);
```

```
            selectionRectangle.setLocation(selectionPoint);
            selectionRectangle.add(e.getPoint());

            // repaints the lines at the old position of the rectangle
            repaintScaled(oldRectangle.x, oldRectangle.y, oldRectangle.width, 1);
            repaintScaled(oldRectangle.x, oldRectangle.y, 1, oldRectangle.height);
            repaintScaled(oldRectangle.x, oldRectangle.y + oldRectangle.height,
                oldRectangle.width, 1);
            repaintScaled(oldRectangle.x + oldRectangle.width, oldRectangle.y, 1,
                oldRectangle.height);

            // repaints the lines of the new rectangle
            repaintScaled(selectionRectangle.x, selectionRectangle.y,
                selectionRectangle.width, 1);
            repaintScaled(selectionRectangle.x, selectionRectangle.y, 1,
                selectionRectangle.height);
            repaintScaled(selectionRectangle.x, selectionRectangle.y +
                selectionRectangle.height, selectionRectangle.width, 1);
            repaintScaled(selectionRectangle.x + selectionRectangle.width,
                selectionRectangle.y, 1, selectionRectangle.height);
        }
    }

    private void updateSizeAndPreferredSize() {
        setSize(drawingArea.width, drawingArea.height);
        setPreferredSize(drawingArea.getSize());
    }

    public void addComponent(AbstractComponent c, boolean atBack) {
        if (componentList.contains(c))
            return;

        /* The order of the following instructions is very important and a bit "
            hacky"
         * since changing them screw the automatic extension of the drawing panel.
         * This is due to state labels extending (and translating) the drawing
             panel before
         * the state is on it.
         */

        if (atBack)
            componentList.addFirst(c);
        else
            componentList.addLast(c);

        c.setDrawingPanel(this);
        extendDrawingArea(c);
        repaintComponent(c);
    }

    public void removeComponent(AbstractComponent c) {
        componentList.remove(c);
        repaintComponent(c);
    }

    public void repaintComponent(AbstractComponent c) {
        repaintScaled(c.getBounds());
    }

    public void repaintScaled(Rectangle r) {
        repaintScaled(r.x, r.y, r.width, r.height);
    }

    public void repaintScaled(int x, int y, int w, int h) {

        //the following code adjusts width and height to prevent too small bounding
            boxes
        //because of a possible "scaling down" here followed by the "scaling up"
```

```
        //in paintComponent

        float invscaling = 1/scaling;
        int wadjust = 0;
        int hadjust = 0;

        if (x % invscaling > 0)
            wadjust++;
        if (w % invscaling > 0)
            wadjust++;
        if (y % invscaling > 0)
            hadjust++;
        if (h % invscaling > 0)
            hadjust++;

        repaint(0, (int)(x*scaling), (int)(y*scaling), (int)(w*scaling)+wadjust, (
            int)(h*scaling)+hadjust);
    }

    public void extendDrawingArea(AbstractComponent c) {
        Rectangle r = c.getBounds();

        r.x *= scaling;
        r.y *= scaling;
        r.width *= scaling;
        r.height *= scaling;

        r.grow(5, 5);

        if (drawingArea.contains(r)) {
            scrollRectToVisible(r);
            return;
        }

        drawingArea.add(r);

        if (drawingArea.x < 0) {
            translate((int)(-drawingArea.x/scaling), 0);
            drawingArea.x = 0;
        }

        if (drawingArea.y < 0) {
            translate(0, (int)(-drawingArea.y/scaling));
            drawingArea.y = 0;
        }

        updateSizeAndPreferredSize();
        scrollRectToVisible(r);
        revalidate();
    }

    private JPopupMenu createContextMenu()  {
        JPopupMenu menu = new JPopupMenu();
        CommonAction[] actions;

        actions = new CommonAction[] {setSelectModeAction, setStateModeAction,
            setTransitionModeAction};

        for (int i = 0; i < actions.length; i++)
            menu.add(actions[i]);

        menu.addSeparator();

        actions = new CommonAction[] {zoomInAction, zoomOutAction};
        for (int i = 0; i < actions.length; i++)
            menu.add(actions[i]);

        menu.addSeparator();
```

```
        menu.add(toggleGridAction);

        return menu;
    }


    private void createActions() {
        setSelectModeAction = new CommonAction("Select mode",
                                               null,
                                               "Edit, move or delete any object",
                                               null,
                                               null) {
                public void actionPerformed(ActionEvent e) {
                    setMode(DrawingPanel.SELECT_MODE);
                }
            };

        setStateModeAction = new CommonAction("State mode",
                                              null,
                                              "Add, edit, move or delete states",
                                              null,
                                              null) {
                public void actionPerformed(ActionEvent e) {
                    setMode(DrawingPanel.STATE_MODE);
                }
            };

        setTransitionModeAction = new CommonAction("Transition mode",
                                                   null,
                                                   "Add, edit or delete transitions
                                                       ",
                                                   null,
                                                   null) {
                public void actionPerformed(ActionEvent e) {
                    setMode(DrawingPanel.TRANSITION_MODE);
                }
            };

        zoomInAction = new CommonAction("Zoom In",
                                        "toolbarButtonGraphics/general/ZoomIn",
                                        "Zoom In",
                                        new Integer(KeyEvent.VK_I),
                                        null) {
                public void actionPerformed(ActionEvent e) {
                    zoomIn();
                }
            };

        zoomOutAction = new CommonAction("Zoom Out",
                                         "toolbarButtonGraphics/general/ZoomOut",
                                         "Zoom Out",
                                         new Integer(KeyEvent.VK_O),
                                         null) {
                public void actionPerformed(ActionEvent e) {
                    zoomOut();
                }
            };

        toggleGridAction = new CommonAction("Toggle grid",
                                            null,
                                            "Toggle the grid",
                                            null,
                                            null) {
                public void actionPerformed(ActionEvent e) {
                    toggleGrid();
                }
            };
```

```
    }

    protected void processEvent(AWTEvent e) {
        if (e instanceof MouseEvent) {
            MouseEvent me = (MouseEvent)e;
            int x = me.getX();
            int y = me.getY();
            int newx = (int) (x / scaling);
            int newy = (int) (y / scaling);
            me.translatePoint(newx - x, newy - y);
        }

        super.processEvent(e);
    }
}
```

C.13 Editable

Listing C.13: Source code for Editable

```
package paxion;

public interface Editable {
    /**
     * Edit this object. Typically this involves popping up a new frame which
     * contains controls to edit the component fields.
     */
    void edit();
}
```

C.14 EditableListPanel

Listing C.14: Source code for EditableListPanel

```
package paxion;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class EditableListPanel extends JPanel {
    protected VectorListModel data;
    protected Class dataType;
    protected JList list;
    protected JButton btNew = createNewButton();
    protected JButton btEdit = createEditButton();
    protected JButton btDelete = createDeleteButton();
    protected JPanel buttonPanel = createButtonPanel();

    public EditableListPanel(String title, VectorListModel v, Class c) {
        super();
        setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));

        data = v;
        dataType = c;

        JLabel label;

        label = new JLabel(title);
```

```
        label.setAlignmentX(Component.CENTER_ALIGNMENT);

        add(label);

        list = new JList(data);
        list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

        JScrollPane scrollPane;

        scrollPane = new JScrollPane(list);

        add(scrollPane);

        add(Box.createRigidArea(new Dimension(0, 5)));

        add(buttonPanel);
    }

    protected JPanel createButtonPanel() {
        JPanel panel;

        panel = new JPanel();
        panel.setLayout(new BoxLayout(panel, BoxLayout.LINE_AXIS));

        // New button
        panel.add(btNew);

        // Empty space
        panel.add(Box.createRigidArea(new Dimension(5, 0)));

        // Edit button
        panel.add(btEdit);

        // Empty space
        panel.add(Box.createRigidArea(new Dimension(5, 0)));

        // Delete button
        panel.add(btDelete);

        return panel;
    }

    protected JButton createNewButton() {
        JButton button;

        button = new JButton("new");
        button.setAlignmentX(Component.CENTER_ALIGNMENT);
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                Editable newValue = null;
                try {
                    newValue = (Editable)dataType.newInstance();
                } catch(Exception exception) {
                    System.err.println(exception.getMessage());
                }

                newValue.edit();
                if (!newValue.toString().equals(""))
                    data.add(newValue);
            }
        });

        return button;
    }

    protected JButton createEditButton() {
        JButton button;
```

```
        button = new JButton("edit");
        button.setAlignmentX(Component.CENTER_ALIGNMENT);
        button.addActionListener(new EditButtonActionListener());

        return button;
    }

    protected JButton createDeleteButton() {
        JButton button;

        button = new JButton("delete");
        button.setAlignmentX(Component.CENTER_ALIGNMENT);
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                data.remove(list.getSelectedValue());
            }
        });

        return button;
    }

    protected class EditButtonActionListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            int index = list.getSelectedIndex();
            if (index != -1) {
                ((Editable)list.getSelectedValue()).edit();
                data.fireContentChanged(index);
            }
        }
    }
}
```

C.15 Editor

Listing C.15: Source code for Editor

```
package paxion;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Editor extends JFrame {
    private CommonAction aboutAction;
    private ProjectEditor projectEditor;
    private Project project;

    public Editor() {
        // Create and set up the window
        super("Editor");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Create a new project
        project = new Project();

        // Create the project editor
        projectEditor = new ProjectEditor(project);

        // Create and set up the menu bar
        setJMenuBar(createMenuBar());

        // Create and set up the content pane
        setContentPane(createContentPane());
```

```
        // Display the window
        pack();
        setSize(new Dimension(800,600));
        setVisible(true);
    }

    public JMenuBar createMenuBar() {
        JMenuBar menuBar = new JMenuBar();
        CommonAction[] actions;
        JMenuItem menuItem;
        JMenu menu;

        // Create the file menu
        menu = new JMenu("File");
        menu.setMnemonic(KeyEvent.VK_F);

        actions = new CommonAction[] {projectEditor.newProjectAction, projectEditor
            .openProjectAction,
                                      projectEditor.saveProjectAction,
                                          projectEditor.saveProjectAsAction,
                                      projectEditor.exportProjectAction};

        for (int i = 0; i < actions.length; i++)
            menu.add(actions[i]);

        menu.addSeparator();

        menuItem = new JMenuItem("Exit");
        menuItem.setMnemonic(KeyEvent.VK_X);
        menuItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_Q, InputEvent.
            CTRL_MASK));
        menuItem.setToolTipText("Exit the program");
        menuItem.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent evt) {
                    System.exit(0);
                }
            });

        menu.add(menuItem);
        menuBar.add(menu);

        // Create the process menu
        menu = new JMenu("Process");
        menu.setMnemonic(KeyEvent.VK_P);

        actions = new CommonAction[] {projectEditor.newProcessAction,
                                      projectEditor.deleteProcessAction};

        for (int i = 0; i < actions.length; i++)
            menu.add(actions[i]);

        menuBar.add(menu);

        // Create the view menu
        menu = new JMenu("View");
        menu.setMnemonic(KeyEvent.VK_V);

        actions = new CommonAction[] {projectEditor.zoomInAction, projectEditor.
            zoomOutAction};
        for (int i = 0; i < actions.length; i++)
            menu.add(actions[i]);

        menuBar.add(menu);

        return menuBar;
    }

    public JToolBar createToolBar() {
```

```
        JToolBar toolBar = new JToolBar();
        JButton button;
        CommonAction[] actions;

        // Add the "file actions" buttons
        actions = new CommonAction[] {projectEditor.newProjectAction, projectEditor
            .openProjectAction,
                                      projectEditor.saveProjectAction,
                                         projectEditor.saveProjectAsAction};
        for (int i = 0; i < actions.length; i++) {
            button = toolBar.add(actions[i]);
            button.setIcon(actions[i].largeIcon);
        }

        // Add a separator
        toolBar.add(new JToolBar.Separator());

        // Add the "view actions" buttons
        actions = new CommonAction[] {projectEditor.zoomInAction, projectEditor.
            zoomOutAction};
        for (int i = 0; i < actions.length; i++) {
            button = toolBar.add(actions[i]);
            button.setIcon(actions[i].largeIcon);
        }

        // Add a separator
        toolBar.add(new JToolBar.Separator());

        // Add the "change mode" buttons
        actions = new CommonAction[] {projectEditor.setSelectModeAction,
            projectEditor.setStateModeAction, projectEditor.setTransitionModeAction
            };
        for (int i = 0; i < actions.length; i++) {
            button = toolBar.add(actions[i]);
            button.setIcon(actions[i].largeIcon);
        }
        return toolBar;
    }

    public Container createContentPane() {
        // Create the content pane
        JPanel contentPane = new JPanel(new BorderLayout());

        // We don't want it transparent
        contentPane.setOpaque(true);

        // Add the tool bar to the content pane.
        contentPane.add(createToolBar(), BorderLayout.NORTH);

        // Add the system editor to the content pane.
        contentPane.add(projectEditor, BorderLayout.CENTER);

        return contentPane;
    }

    public static void main(String[] args) {
        new Editor();
    }
}
```

C.16 Effect

Listing C.16: Source code for Effect

```
package paxion;

import java.io.*;
import java.awt.*;

public class Effect extends AnchorableLabel {
    public Effect(String text) {
        super(text);
    }

    public Effect(String text, Point position) {
        super(text, position);
    }

    public void delete() {
        super.delete();

        ((Transition)parent).setEffect(null);
    }

    public void writeXML(PrintWriter writer) {
        writer.print("<effect ");
        super.writeXML(writer);
        writer.println("</effect>");
    }

    public void writeDivine(PrintWriter writer) {
        writer.print("effect " + text + ";");
    }
}
```

C.17 Guard

Listing C.17: Source code for Guard

```
package paxion;

import java.io.*;
import java.awt.*;

public class Guard extends AnchorableLabel {
    public Guard(String text) {
        super(text);
    }

    public Guard(String text, Point position) {
        super(text, position);
    }

    public void delete() {
        super.delete();

        ((Transition)parent).setGuard(null);
    }

    public void writeXML(PrintWriter writer) {
        writer.print("<guard ");
        super.writeXML(writer);
        writer.println("</guard>");
    }

    public void writeDivine(PrintWriter writer) {
        writer.print("guard " + text + ";");
    }
```

```
}
```

C.18 HasContextMenu

Listing C.18: Source code for HasContextMenu

```
package paxion;

public interface HasContextMenu {
    /**
     * Shows the context menu of the component at position (x, y) in DrawingPanel
     * coordinates.
     *
     * @param x the x coordinate to popup the context menu
     * @param y the y coordinate to popup the context menu
     */
    void showContextMenu(int x, int y);
}
```

C.19 LabelEditor

Listing C.19: Source code for LabelEditor

```
package paxion;

import java.awt.*;
import javax.swing.*;

public class LabelEditor extends ObjectEditor {
    private JTextField tfText;

    public LabelEditor(Frame frame, AnchorableLabel label) {
        super(frame, "Edit Label", label);
    }

    protected JPanel createObjectPropertiesPanel() {
        JPanel panel;

        // Create the content pane
        panel = new JPanel();
        panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
        panel.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));

        tfText = new JTextField(((AnchorableLabel)object).getText());
        tfText.setMaximumSize(new Dimension(Short.MAX_VALUE, Short.MAX_VALUE));

        panel.add(tfText);

        return panel;
    }

    protected void updateObject() {
        ((AnchorableLabel)object).setText(tfText.getText());
    }
}
```

C.20 ObjectEditor

Listing C.20: Source code for ObjectEditor

```
package paxion;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public abstract class ObjectEditor extends JDialog {
    protected Object object;

    public ObjectEditor(Frame frame, String title, Object object) {
        super(frame, title, true);

        this.object = object;

        // Create and set up the window
        setDefaultCloseOperation(JDialog.DISPOSE_ON_CLOSE);

        // Create and set up the content pane
        setContentPane(createContentPane());

        // Display the window
        pack();
        setLocationRelativeTo(frame);
        setVisible(true);
    }

    protected Container createContentPane() {
        JPanel panel;
        JLabel label;
        JButton button;

        // Create the content pane
        JPanel contentPane = new JPanel();
        contentPane.setLayout(new BoxLayout(contentPane, BoxLayout.Y_AXIS));

        // We don't want it transparent
        contentPane.setOpaque(true);

        // Set up the properties panel
        contentPane.add(createObjectPropertiesPanel());

        // Set up the button panel
        panel = new JPanel();
        panel.setLayout(new BoxLayout(panel, BoxLayout.LINE_AXIS));
        panel.setBorder(BorderFactory.createEmptyBorder(0, 10, 10, 10));

        // OK button
        button = new JButton("OK");
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                updateObject();
                closeDialog();
            }
        });
        panel.add(button);

        // Empty space
        panel.add(Box.createRigidArea(new Dimension(10, 0)));

        // Cancel button
        button = new JButton("Cancel");
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
```

```
                closeDialog();
            }
        });
        panel.add(button);

        contentPane.add(panel);

        return contentPane;
    }

    protected void closeDialog() {
        setVisible(false);
        dispose();
    }

    abstract protected JPanel createObjectPropertiesPanel();
    abstract protected void updateObject();
}
```

C.21 ProcessEditor

Listing C.21: Source code for ProcessEditor

```
package paxion;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class ProcessEditor extends JPanel {
    // drawingScrollPane (containing the drawing panel) initial preferred size
    private static final Dimension SCROLLPANE_INITIAL_SIZE = new Dimension(500,
        500);

    private DrawingPanel drawingPanel;
    private JScrollPane drawingScrollPane;

    private JTextField tfName;

    private Process process;
    private ProjectEditor projectEditor;

    public ProcessEditor(ProjectEditor pe, Process p) {
        super();

        process = p;
        projectEditor = pe;

        setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));

        // Setup the drawing panel
        drawingPanel = new DrawingPanel(p);

        // Put the drawing panel in a scroll pane
        drawingScrollPane = new JScrollPane(drawingPanel);
        drawingScrollPane.setPreferredSize(SCROLLPANE_INITIAL_SIZE);

        // Setup the properties pane
        JPanel propertiesPane = new JPanel();
        propertiesPane.setLayout(new BoxLayout(propertiesPane, BoxLayout.Y_AXIS));
        propertiesPane.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));

        JPanel panel;
```

```
        JLabel label;
        JButton button;

        // name panel
        panel = new JPanel();
        panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));

        label = new JLabel("name");
        label.setAlignmentX(Component.CENTER_ALIGNMENT);
        panel.add(label);

        tfName = new JTextField(process.getName());
        tfName.setMaximumSize(new Dimension(Short.MAX_VALUE, (int)tfName.
            getPreferredSize().getHeight()));
        tfName.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                String name = tfName.getText();
                process.setName(name);
                projectEditor.setProcessName(ProcessEditor.this, name);
            }
        });
        panel.add(tfName);

        propertiesPane.add(panel);

        // empty space
        propertiesPane.add(Box.createRigidArea(new Dimension(0, 10)));

        // variables panel
        panel = new EditableListPanel("variables", process.getVariables(), Variable
            .class);
        propertiesPane.add(panel);

        // empty space
        propertiesPane.add(Box.createRigidArea(new Dimension(0, 10)));

        // acceptation sets panel
        panel = new AcceptStatesSetListPanel("acceptation sets", process.
            getAcceptStateSets());
        propertiesPane.add(panel);

        //
        propertiesPane.setPreferredSize(propertiesPane.getMinimumSize());

        // Create the split pane
        JSplitPane splitPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
            drawingScrollPane, propertiesPane);
        splitPane.setOneTouchExpandable(true);
        splitPane.setResizeWeight(1.0);

        // Add the split pane to the panel
        add(splitPane);
    }

    //---------------

    public Process getProcess() {
        return process;
    }

    public DrawingPanel getDrawingPanel() {
        return drawingPanel;
    }

    public void update() {
        drawingPanel.update();
    }
}
```

C.22 Process

Listing C.22: Source code for Process

```
package paxion;

import java.io.*;
import java.util.*;

public class Process {
    private static int nextProcessId = 0;

    private Collection transitions = new HashSet();
    private Collection states = new HashSet();
    private VectorListModel variables = new VectorListModel();
    private VectorListModel acceptStateSets = new VectorListModel();

    private State initialState = null;
    private String name;
    private int id;

    public Process() {
        this(nextProcessId, "Process" + (nextProcessId + 1));
    }

    public Process(int id, String name) {
        this.name = name;
        this.id = id;
        if (id >= nextProcessId)
            nextProcessId = id + 1;
    }

    public int getId() {
        return id;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void addState(State s) {
        states.add(s);
    }

    public State findState(int id) {
        Iterator i = states.iterator();

        while (i.hasNext()) {
            State s = (State)i.next();
            if (s.getId() == id)
                return s;
        }

        return null;
    }

    public void removeState(State s) {
        Iterator i;
```

```
        // remove all transitions involving this state
        i = s.getTransitions().iterator();
        while (i.hasNext())
            transitions.remove(i.next());

        // remove the state from all the accepting states sets it is in
        i = acceptStateSets.iterator();
        while (i.hasNext())
            ((AcceptStatesSet)i.next()).removeState(s);

        // remove the state from the state list
        states.remove(s);

        // if that state was the initial state... well... we have no more initial
            state...
        if (initialState == s)
            initialState = null;
    }

    public Collection getStates() {
        return states;
    }

    public void addTransition(Transition t) {
        transitions.add(t);
    }

    public void removeTransition(Transition t) {
        t.getOrigin().removeTransition(t);
        t.getDestination().removeTransition(t);
        transitions.remove(t);
    }

    public Collection getTransitions() {
        return transitions;
    }

    public void addVariable(Variable v) {
        variables.add(v);
    }

    public VectorListModel getVariables() {
        return variables;
    }

    public void addAcceptStateSet(AcceptStatesSet a) {
        acceptStateSets.add(a);
    }

    public VectorListModel getAcceptStateSets() {
        return acceptStateSets;
    }

    public void setInitialStateStatus(State s, boolean b) {
        if ((initialState != null) && (b == true))
            initialState.setInitial(false);

        s.setInitial(b);
        initialState = s;
    }

    public void writeXML(PrintWriter writer) {
        Iterator i;

        writer.println("<process id=\"" + id + "\" name=\"" + name + "\">");

        i = variables.iterator();
```

```
        while (i.hasNext())
            ((Variable)i.next()).writeXML(writer);

        i = states.iterator();
        while (i.hasNext())
            ((State)i.next()).writeXML(writer);

        if (initialState == null)
            System.err.println("Warning: no initial state!");
        else
            writer.println("<initstate id=\"" + initialState.getId() + "\"/>");

        i = acceptStateSets.iterator();
        while (i.hasNext())
            ((AcceptStatesSet)i.next()).writeXML(writer);

        i = transitions.iterator();
        while (i.hasNext())
            ((Transition)i.next()).writeXML(writer);

        writer.println("</process>");
    }

    public void writeDivine(PrintWriter writer) throws NoInitialStateException {
        Iterator i;

        if (initialState == null)
            throw new NoInitialStateException();

        writer.println("process " + name + " {");

        // Iterate through all possible variable types
        for (int t = 0; t < Variable.numberTypes; t++) {

            // Write all variables of current type on one line
            int c = 0;
            i = variables.iterator();
            while (i.hasNext()) {
                Variable v = (Variable)i.next();

                if (v.getType() == t) {

                    // If it is the first variable of this type, we have to write
                        the type string
                    if (c == 0)
                        writer.print(Variable.typeString[t] + " ");
                    else
                        writer.print(", ");

                    v.writeDivine(writer);

                    c++;
                }
            }

            if (c > 0)
                writer.println(";");
        }

        writer.print("state ");
        i = states.iterator();
        while (i.hasNext()) {
            ((State)i.next()).writeDivine(writer);
            if (i.hasNext())
                writer.print(", ");
        }
        writer.println(";");
```

```
        writer.print("init ");
        initialState.writeDivine(writer);
        writer.println(";");

        i = acceptStateSets.iterator();
        while (i.hasNext()) {
            ((AcceptStatesSet)i.next()).writeDivine(writer);

            if (!i.hasNext())
                writer.print("\n");
        }

        writer.println("trans ");
        i = transitions.iterator();
        while (i.hasNext()) {
            ((Transition)i.next()).writeDivine(writer);
            if (i.hasNext())
                writer.println(", ");
        }
        writer.println(";\n}\n");
    }

    public class NoInitialStateException extends Exception {
        public NoInitialStateException() {
            super("Process " + name + " has no initial state");
        }
    }
}
```

C.23 ProjectEditor

Listing C.23: Source code for ProjectEditor

```
package paxion;

import java.io.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;
import javax.swing.filechooser.FileFilter;
import org.xml.sax.*;
import org.xml.sax.helpers.*;

public class ProjectEditor extends JPanel {
    public CommonAction newProjectAction, openProjectAction, saveProjectAction,
        saveProjectAsAction, exportProjectAction;
    public CommonAction newProcessAction, openProcessAction, saveProcessAsAction,
        deleteProcessAction;
    public CommonAction setSelectModeAction, setStateModeAction,
        setTransitionModeAction, zoomInAction, zoomOutAction;

    private JLabel statusLabel;

    private JComboBox cbType;
    private JComboBox cbObserver;

    private JTabbedPane tabbedPane;

    private JFileChooser fc = new JFileChooser();
    private XMLReader parser;
    private File currentProject = null;
    private Project project;
```

```
    private FileFilter divineFileFilter = new DivineFileFilter();
    private FileFilter xmlFileFilter = new XMLFileFilter();

    public ProjectEditor(Project p) {
        super(new BorderLayout());

        project = p;

        // We don't want the panel to be transparent
        setOpaque(true);

        // Create the tabbed pane
        tabbedPane = new JTabbedPane();

        // Create the properties pane
        JPanel propertiesPane = new JPanel();
        propertiesPane.setLayout(new BoxLayout(propertiesPane, BoxLayout.Y_AXIS));
        propertiesPane.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));

        JPanel panel;
        JLabel label;

        label = new JLabel("type");
        label.setAlignmentX(Component.CENTER_ALIGNMENT);
        propertiesPane.add(label);

        // project type
        panel = new JPanel();
        panel.setLayout(new BoxLayout(panel, BoxLayout.LINE_AXIS));

        cbType = new JComboBox(Project.typeString);
        cbType.setMaximumSize(new Dimension((int)cbType.getPreferredSize().getWidth
            (), (int)cbType.getPreferredSize().getHeight()));
        cbType.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                int type = cbType.getSelectedIndex();
                project.setType(type);
                setType(type);
            }
        });

        panel.add(cbType);

        panel.add(Box.createRigidArea(new Dimension(10, 0)));

        label = new JLabel("observer");
        panel.add(label);

        panel.add(Box.createRigidArea(new Dimension(5, 0)));

        String[] observers = { "none" };

        cbObserver = new JComboBox(observers);
        cbObserver.setMaximumSize(new Dimension(Short.MAX_VALUE, (int)cbObserver.
            getPreferredSize().getHeight()));
        cbObserver.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                int observerNumber = cbObserver.getSelectedIndex();
                Process observer;

                if (observerNumber > 0)
                    observer = ((ProcessEditor)tabbedPane.getComponentAt(
                        observerNumber - 1)).getProcess();
                else {
                    observer = null;
                }

                project.setObserver(observer);
```

```
        }
    });
    panel.add(cbObserver);

    propertiesPane.add(panel);

    // empty space
    propertiesPane.add(Box.createRigidArea(new Dimension(0, 10)));

    // variables
    panel = new EditableListPanel("variables", project.getVariables(), Variable
        .class);
    propertiesPane.add(panel);

    // empty space
    propertiesPane.add(Box.createRigidArea(new Dimension(0, 10)));

    // channels
    panel = new EditableListPanel("channels", project.getChannels(), Channel.
        class);
    propertiesPane.add(panel);

    // set the preferred size of the properties panel to its minimum size
    propertiesPane.setPreferredSize(propertiesPane.getMinimumSize());

    // Create the split pane
    JSplitPane splitPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
        propertiesPane, tabbedPane);
    splitPane.setOneTouchExpandable(true);

    // hide the properties panel
    splitPane.setDividerLocation(0.0);

    // Add the split pane to the panel
    add(splitPane, BorderLayout.CENTER);

    // Set up the status label
    statusLabel = new JLabel("ready");

    // Add the status label to the panel
    add(statusLabel, BorderLayout.SOUTH);

    //----------------
    // Init parser
    ProjectParserDataHandler dataHandler = new ProjectParserDataHandler(project
        );

    final String VALIDATION_FEATURE_ID = "http://xml.org/sax/features/
        validation";
    try {
        parser = XMLReaderFactory.createXMLReader("org.apache.xerces.parsers.
            SAXParser");

        // This activates the validation feature of Xerces. It means that the
            XML parser
        // will check if the XML file is conform to the DTD and will throw an
        // exception (and stop parsing) if it's not.
        parser.setFeature(VALIDATION_FEATURE_ID, true);
    } catch (SAXException e) {
        System.err.println(e.getMessage());
    }

    parser.setEntityResolver(dataHandler);
    parser.setContentHandler(dataHandler);
    parser.setErrorHandler(dataHandler);
    parser.setDTDHandler(dataHandler);

    // Create the actions
```

```
        createActions();

        // Create a new (empty) process and add it to the project
        addEmptyProcess();
    }

    public void setProcessName(int index, String name) {
        // Update tab title
        tabbedPane.setTitleAt(index, name);

        // Update observer combo box
        int observerIndex = cbObserver.getSelectedIndex();
        cbObserver.removeItemAt(index + 1);
        cbObserver.insertItemAt(name, index + 1);
        cbObserver.setSelectedIndex(observerIndex);
    }

    public void setProcessName(ProcessEditor pe, String name) {
        setProcessName(tabbedPane.indexOfComponent(pe), name);
    }

    private void createActions() {
        newProjectAction = new CommonAction("New",
                                            "toolbarButtonGraphics/general/New",
                                            "Create a new project",
                                            new Integer(KeyEvent.VK_N),
                                            KeyStroke.getKeyStroke(KeyEvent.VK_N,
                                                InputEvent.CTRL_MASK)) {
            public void actionPerformed(ActionEvent e) {
                newProject();
            }
        };

        openProjectAction = new CommonAction("Open",
                                             "toolbarButtonGraphics/general/Open",
                                             "Open an existing project",
                                             new Integer(KeyEvent.VK_O),
                                             KeyStroke.getKeyStroke(KeyEvent.VK_O,
                                                 InputEvent.CTRL_MASK)) {
            public void actionPerformed(ActionEvent e) {
                openProject();
            }
        };

        saveProjectAction = new CommonAction("Save",
                                             "toolbarButtonGraphics/general/Save",
                                             "Save current project",
                                             new Integer(KeyEvent.VK_S),
                                             KeyStroke.getKeyStroke(KeyEvent.VK_S,
                                                 InputEvent.CTRL_MASK)) {
            public void actionPerformed(ActionEvent e) {
                saveProject();
            }
        };

        saveProjectAsAction = new CommonAction("SaveAs",
                                               "toolbarButtonGraphics/general/
                                                   SaveAs",
                                               "Save current project as...",
                                               new Integer(KeyEvent.VK_A),
                                               null) {
            public void actionPerformed(ActionEvent e) {
                saveProjectAs();
            }
        };

        exportProjectAction = new CommonAction("Export",
                                               null,
```

```
                                        "Export current project",
                                        new Integer(KeyEvent.VK_E),
                                        null) {
            public void actionPerformed(ActionEvent e) {
                exportProject();
            }
        };

        // ----------

        newProcessAction = new CommonAction("New",
                                        "toolbarButtonGraphics/general/New",
                                        "Create a new process",
                                        new Integer(KeyEvent.VK_N),
                                        null) {
            public void actionPerformed(ActionEvent e) {
                newProcess();
            }
        };

        deleteProcessAction = new CommonAction("Delete",
                                        null,
                                        "Delete current process",
                                        new Integer(KeyEvent.VK_C),
                                        KeyStroke.getKeyStroke(KeyEvent.VK_W,
                                            InputEvent.CTRL_MASK)) {
            public void actionPerformed(ActionEvent e) {
                deleteProcess();
            }
        };

        // ----------

        setSelectModeAction = new CommonAction("Select",
                                        null,
                                        "Select objects",
                                        null,
                                        null) {
            public void actionPerformed(ActionEvent e) {
                getCurrentDrawingPanel().setMode(DrawingPanel.SELECT_MODE);
            }
        };

        setStateModeAction = new CommonAction("States",
                                        null,
                                        "Place states",
                                        null,
                                        null) {
            public void actionPerformed(ActionEvent e) {
                getCurrentDrawingPanel().setMode(DrawingPanel.STATE_MODE);
            }
        };

        setTransitionModeAction = new CommonAction("Transitions",
                                            null,
                                            "Place transitions",
                                            null,
                                            null) {
            public void actionPerformed(ActionEvent e) {
                getCurrentDrawingPanel().setMode(DrawingPanel.TRANSITION_MODE);
            }
        };

        zoomInAction = new CommonAction("Zoom In",
                                    "toolbarButtonGraphics/general/ZoomIn",
                                    "Zoom In",
                                    new Integer(KeyEvent.VK_I),
                                    null) {
```

```
            public void actionPerformed(ActionEvent e) {
                getCurrentDrawingPanel().zoomIn();
            }
        };

    zoomOutAction = new CommonAction("Zoom Out",
                                     "toolbarButtonGraphics/general/ZoomOut",
                                     "Zoom Out",
                                     new Integer(KeyEvent.VK_O),
                                     null) {
            public void actionPerformed(ActionEvent e) {
                getCurrentDrawingPanel().zoomOut();
            }
        };
}

private boolean openProject(File f) {
    boolean success = true;

    clearProject();

    statusLabel.setText("Opening project: " + f.getName() + "... ");

    try {
        parser.parse(new InputSource(new FileInputStream(f)));
    } catch (Exception e) {
        System.err.println("Warning: open project failed: " + e);
        success = false;
    }

    setType(project.getType());

    tabbedPane.removeAll();

    Process observer = project.getObserver();

    if (observer == null)
        setObserver(0);

    Iterator i = project.getProcesses().iterator();
    while (i.hasNext()) {
        Process p = (Process)i.next();
        addProcess(p);

        if (p == observer)
            // Select last process added as observer process
            setObserver(cbObserver.getItemCount() - 1);
    }

    if (success)
        statusLabel.setText("Opened project successfully.");
    else
        statusLabel.setText("Open project failed!");

    return success;
}

private boolean saveProject(File f) {
    boolean success = true;
    PrintWriter writer;

    statusLabel.setText("Saving project to: " + f.getName() + "... ");

    try {
        writer = new PrintWriter(new FileWriter(f));
        writer.println("<?xml version=\"1.0\" encoding=\"UTF-8\"?>");
        writer.println("<!DOCTYPE system SYSTEM \"data/paxion.dtd\">");
        // We could also use this form:
```

```
//          writer.println("<!DOCTYPE system PUBLIC \"[PubidLiteral]\" \"[URI]\">")
    ;
            project.writeXML(writer);
            writer.close();
        } catch (Exception e) {
            System.err.println("Warning: save project failed: " + e);
            success = false;
        }

        if (success)
            statusLabel.setText("Save successful.");
        else
            statusLabel.setText("Save failed!");

        return success;
    }

    private boolean exportProject(File f) {
        boolean success = true;
        PrintWriter writer;

        statusLabel.setText("Exporting project to: " + f.getName() + "... ");

        try {
            writer = new PrintWriter(new FileWriter(f));
            project.writeDivine(writer);
            writer.close();
        } catch (Exception e) {
            statusLabel.setText("Export failed: "+ e.getMessage());
            success = false;
        }

        if (success)
            statusLabel.setText("Export successful.");

        return success;
    }

    private void newProject() {
        clearProject();
        addEmptyProcess();
        setType(0);
        currentProject = null;

        statusLabel.setText("New project created.");
    }

    private void openProject() {
        int returnVal;

        fc.addChoosableFileFilter(xmlFileFilter);

        returnVal = fc.showOpenDialog(ProjectEditor.this);
        if (returnVal == JFileChooser.APPROVE_OPTION) {
            File f = fc.getSelectedFile();
            if (openProject(f))
                currentProject = f;
        } else {
            statusLabel.setText("Open project action cancelled by user.");
        }

        fc.resetChoosableFileFilters();
    }

    private void saveProject() {
        if (currentProject != null)
            saveProject(currentProject);
        else
```

```
            saveProjectAs();
    }

    private void saveProjectAs() {
        int returnVal;

        fc.addChoosableFileFilter(xmlFileFilter);

        returnVal = fc.showSaveDialog(this);
        if (returnVal == JFileChooser.APPROVE_OPTION) {
            File f = fc.getSelectedFile();
            if (saveProject(f))
                currentProject = f;
        }

        fc.resetChoosableFileFilters();
    }

    private void exportProject() {
        int returnVal;

        fc.addChoosableFileFilter(divineFileFilter);

        returnVal = fc.showDialog(this, "Export");
        if (returnVal == JFileChooser.APPROVE_OPTION) {
            File f = fc.getSelectedFile();
            exportProject(f);
        }

        fc.resetChoosableFileFilters();
    }

    // ----------

    private void newProcess() {
        addEmptyProcess();
        statusLabel.setText("New process created.");
    }

    private void deleteProcess() {
        int index = tabbedPane.getSelectedIndex();

        ProcessEditor pe = (ProcessEditor)tabbedPane.getComponentAt(index);

        project.removeProcess(pe.getProcess());

        tabbedPane.remove(index);
        cbObserver.removeItemAt(index + 1);

        if (tabbedPane.getTabCount() == 0)
            addEmptyProcess();
    }

    // ----------

    private void clearProject() {
        project.clear();

        tabbedPane.removeAll();

        cbObserver.removeAllItems();
        cbObserver.addItem("none");
    }

    private void addEmptyProcess() {
        // Create a new (empty) process
        Process p = new Process();
```

```
        // Add it to the project
        project.addProcess(p);

        // Add it to the editor
        addProcess(p);
    }

    private void addProcess(Process process) {
        // Create a new process editor for it
        ProcessEditor processEditor = new ProcessEditor(this, process);

        // Add it to the tabbed pane
        tabbedPane.addTab(process.getName(), processEditor);

        // Add it to the observer combo box
        cbObserver.addItem(process.getName());

        /* Update the drawing panel of the processEditor. This can't be done in the
         * constructor of the drawing panel because for the update we need the
           graphic
         * context (among others to determine label sizes) and it is only available
           when
         * the "parent chain" is complete (that is after the processEditor has been
           added
         * to the tabbedPane).
         */
        processEditor.update();
    }

    private void setType(int type) {
        cbType.setSelectedIndex(type);

        if (type == 0)
            cbObserver.setEnabled(true);
        else if (type == 1)
            cbObserver.setEnabled(false);
    }

    private void setObserver(int observerNumber) {
        cbObserver.setSelectedIndex(observerNumber);
    }

    private ProcessEditor getCurrentProcessEditor() {
        return ((ProcessEditor)tabbedPane.getSelectedComponent());
    }

    private DrawingPanel getCurrentDrawingPanel() {
        return getCurrentProcessEditor().getDrawingPanel();
    }
}
```

C.24 Project

Listing C.24: Source code for Project

```
package paxion;

import java.io.*;
import java.util.*;

public class Project {
    public static final String[] typeString = { "async", "sync" };

    private VectorListModel variables = new VectorListModel();
```

```
    private VectorListModel channels = new VectorListModel();
    private Collection processes = new LinkedHashSet();

    private Process observer = null;
    private int type = 0;

    public Project() {
    }

    public void clear() {
        variables.clear();
        channels.clear();
        processes.clear();

        observer = null;
        type = 0;
    }

    public void setType(int type) {
        this.type = type;
    }

    public int getType() {
        return type;
    }

    public void setObserver(Process observer) {
        this.observer = observer;
    }

    public Process getObserver() {
        return observer;
    }

    public void addProcess(Process p) {
        processes.add(p);
    }

    public Process findProcess(int id) {
        Iterator i = processes.iterator();

        while (i.hasNext()) {
            Process p = (Process)i.next();
            if (p.getId() == id)
                return p;
        }

        return null;
    }

    public void removeProcess(Process p) {
        if (observer == p)
            observer = null;

        processes.remove(p);
    }

    public void addVariable(Variable v) {
        variables.add(v);
    }

    public void removeVariable(Variable v) {
        variables.remove(v);
    }

    public void addChannel(Channel c) {
        channels.add(c);
    }
```

```
public void removeVariable(Channel c) {
    channels.remove(c);
}

public VectorListModel getChannels() {
    return channels;
}

public VectorListModel getVariables() {
    return variables;
}

public Collection getProcesses() {
    return processes;
}

public void writeXML(PrintWriter writer) {
    Iterator i;

    writer.println("<system type=\"" + type + "\">");

    i = channels.iterator();
    while (i.hasNext())
        ((Channel)i.next()).writeXML(writer);

    i = variables.iterator();
    while (i.hasNext())
        ((Variable)i.next()).writeXML(writer);

    i = processes.iterator();
    while (i.hasNext())
        ((Process)i.next()).writeXML(writer);

    if (observer != null)
        writer.println("<observer id=\"" + observer.getId() + "\"/>");

    writer.println("</system>");
}

public void writeDivine(PrintWriter writer) throws Process.
    NoInitialStateException {
    Iterator i;

    if (channels.size() > 0) {
        writer.print("channel ");

        i = channels.iterator();
        while (i.hasNext()) {
            ((Channel)i.next()).writeDivine(writer);

            if (i.hasNext())
                writer.print(", ");
        }

        writer.println(";");
    }

    // Iterate through all possible variable types
    for (int t = 0; t < Variable.numberTypes; t++) {

        // Write all variables of current type on one line
        int c = 0;
        i = variables.iterator();
        while (i.hasNext()) {
            Variable v = (Variable)i.next();

            if (v.getType() == t) {
```

```
                    // If it is the first variable of this type, we have to write
                       the type string
                    if (c == 0)
                        writer.print(Variable.typeString[t] + " ");
                    else
                        writer.print(", ");

                    v.writeDivine(writer);

                    c++;
                }
            }

            if (c > 0)
                writer.println(";");
        }

        i = processes.iterator();
        while (i.hasNext())
            ((Process)i.next()).writeDivine(writer);

        String observerString;

        if (observer == null)
            observerString = "";
        else
            observerString = " property " + observer.getName();

        writer.println("system " + typeString[type] + observerString + ";");
    }
}
```

C.25 ProjectParserDataHandler

Listing C.25: Source code for ProjectParserDataHandler

```
package paxion;

import java.util.*;
import java.awt.Point;

import org.xml.sax.*;
import org.xml.sax.helpers.*;

public class ProjectParserDataHandler extends DefaultHandler {
    private Project project;
    private LinkedList parentObjects = new LinkedList();

    private Object parentObject = null;
    private Object currentObject;

    public ProjectParserDataHandler(Project p) {
        project = p;
    }

    public InputSource resolveEntity(String publicId, String systemId)
        throws SAXException {
        return null;
    }

    // DTD handling functions

    // Receive notification of a notation declaration event.
```

```
public void notationDecl(String name, String publicId, String systemId) {
}

// Receive notification of an unparsed entity declaration event.
public void unparsedEntityDecl(String name, String publicId, String systemId,
    String notationName) {
}

// normal data handling functions

public void startDocument() {
}

public void startElement(String uri, String localName, String qName, Attributes
     atts) {
    if (parentObjects.size() > 0)
        parentObject = parentObjects.getLast();
    else
        parentObject = null;

    if (localName.equals("point")) {

        currentObject = new Point(Integer.parseInt(atts.getValue("x")),
                                  Integer.parseInt(atts.getValue("y")));

        if (parentObject instanceof AnchorableLabel)
            ((AnchorableLabel)parentObject).setDeltaToAnchorPoint((Point)
                currentObject);
        else if (parentObject instanceof CenteredComponent)
            ((CenteredComponent)parentObject).setCenterPoint((Point)
                currentObject);
        else if (parentObject instanceof Transition)
            ((Transition)parentObject).insertPoint((Point)currentObject);

    } else if (localName.equals("label")) {
        currentObject = new StateLabel(atts.getValue("text"));

        if (parentObject instanceof State)
            ((State)parentObject).setLabel((StateLabel)currentObject);

    } else if (localName.equals("guard")) {
        currentObject = new Guard(atts.getValue("text"));

        if (parentObject instanceof Transition)
            ((Transition)parentObject).setGuard((Guard)currentObject);

    } else if (localName.equals("synchro")) {
        currentObject = new Synchro(atts.getValue("text"));

        if (parentObject instanceof Transition)
            ((Transition)parentObject).setSynchro((Synchro)currentObject);

    } else if (localName.equals("effect")) {
        currentObject = new Effect(atts.getValue("text"));

        if (parentObject instanceof Transition)
            ((Transition)parentObject).setEffect((Effect)currentObject);

    } else if (localName.equals("transition")) {
        int fromStateId = Integer.parseInt(atts.getValue("from"));
        int toStateId = Integer.parseInt(atts.getValue("to"));

        State fromState = ((Process)parentObject).findState(fromStateId);
        State toState = ((Process)parentObject).findState(toStateId);

        currentObject = new Transition(fromState, toState);

        if (parentObject instanceof Process)
```

```
            ((Process)parentObject).addTransition((Transition)currentObject);

        } else if (localName.equals("state")) {
            currentObject = new State(Integer.parseInt(atts.getValue("id")));

            if (parentObject instanceof Process)
                ((Process)parentObject).addState((State)currentObject);

        } else if (localName.equals("initstate")) {
            int stateId = Integer.parseInt(atts.getValue("id"));
            State initialState = ((Process)parentObject).findState(stateId);

            currentObject = null;

            if (parentObject instanceof Process)
                ((Process)parentObject).setInitialStateStatus(initialState, true);

        } else if (localName.equals("acceptset")) {
            int setId = Integer.parseInt(atts.getValue("id"));

            currentObject = new AcceptStatesSet(setId);

            if (parentObject instanceof Process)
                ((Process)parentObject).addAcceptStateSet((AcceptStatesSet)
                    currentObject);

        } else if (localName.equals("variable")) {
            currentObject = new Variable(Integer.parseInt(atts.getValue("type")),
                atts.getValue("name"));

            String s;

            s = atts.getValue("length");
            if (s != null)
                ((Variable)currentObject).setLength(Integer.parseInt(s));

            s = atts.getValue("initialValue");
            if (s != null)
                ((Variable)currentObject).setInitialValue(Integer.parseInt(s));

            if (parentObject instanceof Process)
                ((Process)parentObject).addVariable((Variable)currentObject);
            else if (parentObject instanceof Project)
                ((Project)parentObject).addVariable((Variable)currentObject);

        } else if (localName.equals("channel")) {
            currentObject = new Channel(atts.getValue("name"));

            if (parentObject instanceof Project)
                ((Project)parentObject).addChannel((Channel)currentObject);

        } else if (localName.equals("process")) {
            currentObject = new Process(Integer.parseInt(atts.getValue("id")), atts
                .getValue("name"));

            if (parentObject instanceof Project)
                ((Project)parentObject).addProcess((Process)currentObject);

        } else if (localName.equals("observer")) {
            currentObject = null;

            int processId = Integer.parseInt(atts.getValue("id"));

            if (parentObject instanceof Project) {
                Process observer = ((Project)parentObject).findProcess(processId);
                ((Project)parentObject).setObserver(observer);
            }
```

```
        } else if (localName.equals("system")) {
            currentObject = project;

            project.setType(Integer.parseInt(atts.getValue("type")));
        } else {
            System.err.println("Warning: invalid tag found (" + localName + ") !");
        }

        parentObjects.addLast(currentObject);
    }

    public void characters(char[] ch, int start, int length) {
        String s = new String(ch, start, length);

        if (s.equals("\n"))
            return;

        if (currentObject instanceof AcceptStatesSet) {
            String[] stateIdStrings = s.split(":");

            for (int i = 0; i < stateIdStrings.length; i++) {
                int stateId = Integer.parseInt(stateIdStrings[i]);
                State state = ((Process)parentObject).findState(stateId);
                ((AcceptStatesSet)currentObject).addState(state);
            }
        } else
            System.err.println("Warning: character data (" + s + ") found in
                invalid tag !");
    }

    public void endElement(String uri, String localName, String qName) {
        parentObjects.removeLast();
    }

    public void endDocument() {
    }

    // Receive notification of a warning.
    public void warning(SAXParseException e) throws SAXException {
        throw e;
    }

    // Receive notification of a recoverable error.
    public void error(SAXParseException e) throws SAXException {
        throw e;
    }
}
```

C.26 Selectable

Listing C.26: Source code for Selectable

```
package paxion;

import java.awt.Color;

public interface Selectable {
    public static final Color DEFAULT_NORMAL_COLOR = Color.BLACK;
    public static final Color DEFAULT_SELECTED_COLOR = Color.ORANGE;

    /**
     * Changes selected status of this component. This will typically involve
     * a visual change.
     *
```

```
     * @param selected the new selected status
     */
    void setSelected(boolean selected);

    /**
     * Returns wether this component is currently selected or not.
     *
     * @return the current selected status for this component
     */
    boolean isSelected();
}
```

C.27 StateEditor

Listing C.27: Source code for StateEditor

```
package paxion;

import java.awt.*;
import javax.swing.*;

public class StateEditor extends ObjectEditor {
    private JTextField tfName;
    private JCheckBox cbInitial;

    public StateEditor(Frame frame, State state) {
        super(frame, "Edit State", state);
    }

    protected JPanel createObjectPropertiesPanel() {
        JPanel propertiesPanel;
        JPanel panel;
        JLabel label;

        // Create the properties panel
        propertiesPanel = new JPanel();
        propertiesPanel.setLayout(new BoxLayout(propertiesPanel, BoxLayout.Y_AXIS))
            ;
        propertiesPanel.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));

        panel = new JPanel();
        panel.setLayout(new BoxLayout(panel, BoxLayout.LINE_AXIS));

        label = new JLabel("name:");
        tfName = new JTextField(((State)object).getName());
        tfName.setMaximumSize(new Dimension(Short.MAX_VALUE, Short.MAX_VALUE));

        panel.add(label);
        panel.add(Box.createRigidArea(new Dimension(10, 0)));
        panel.add(tfName);

        propertiesPanel.add(panel);

        propertiesPanel.add(Box.createRigidArea(new Dimension(0, 10)));

        cbInitial = new JCheckBox("initial state", ((State)object).isInitial());

        propertiesPanel.add(cbInitial);

        return propertiesPanel;
    }

    protected void updateObject() {
        ((State)object).setName(tfName.getText());
```

```
        ((State)object).drawingPanel.getProcess().setInitialStateStatus((State)
            object, cbInitial.isSelected());
    }
}
```

C.28 State

Listing C.28: Source code for State

```
package paxion;

import java.io.*;
import java.util.*;
import java.awt.*;
import java.awt.geom.*;

public class State extends CenteredComponent {
    // —— public constants ——

    public static final float RADIUS = 15.0f;
    public static final float DIAMETER = RADIUS*2;
    public static final int DEFAULT_LABEL_POSITION_X = 0;
    public static final int DEFAULT_LABEL_POSITION_Y = -(int)DIAMETER;
    public static final Color DEFAULT_FILL_COLOR = new Color(0.7f, 0.2f, 0.1f);

    // —— public static variables ——

    // "replaces" AbstractComponent's normalColor
    public static Color fillColor = DEFAULT_FILL_COLOR;

    // —— private constants ——

    private static final Color CONTOUR_COLOR = Color.BLACK;
    private static final BasicStroke STROKE = new BasicStroke(1.0f);

    // —— private static variables ——

    private static int nextStateId = 0;

    // —— private variables ——

    private Ellipse2D.Float circle = new Ellipse2D.Float((float)1.0f, (float)1.0f,
        DIAMETER, DIAMETER);
    private Collection transitions = new HashSet();
    private StateLabel label = null;

    private boolean initial = false;
    private int id;

    // This is needed to prevent concurrent modification of the transitions set
    private boolean deletingState = false;

    // —— public static methods ——

    public static void setFillColor(Color c) {
        fillColor = c;
    }


    // this constructor is used when loading from XML
    public State(int id) {
        this(id, 0, 0, null);
    }
```

```
public State(int x, int y) {
    this(nextStateId, x, y, null);

    setName("State" + nextStateId);
}

public State(int id, int x, int y, StateLabel label) {
    setSize((int)DIAMETER+2, (int)DIAMETER+2);
    setDeltaToCenter((int)RADIUS+1, (int)RADIUS+1);
    setCenter(x, y);
    setLabel(label);

    this.id = id;
    if (id >= nextStateId)
        nextStateId = id + 1;

    currentColor = fillColor;
}

public int getId() {
    return id;
}

public Color getNormalColor() {
    return fillColor;
}

public void setLabel(StateLabel l) {
    label = l;

    if (label != null) {
        label.setParent(this);
        label.setAnchorPoint(center);

        if (drawingPanel != null)
            drawingPanel.addLabel(label);
    }
}

public AnchorableLabel getLabel() {
    return label;
}

public void setName(String name) {
    if (label == null) {
        if (!name.equals(""))
            setLabel(
                new StateLabel(
                    name,
                    new Point(DEFAULT_LABEL_POSITION_X,
                        DEFAULT_LABEL_POSITION_Y)));
    } else
        label.setText(name);
}

public String getName() {
    if (label == null) {
        return "";
    } else
        return label.getText();
}

public void setInitial(boolean b) {
    initial = b;
    repaint();
}

public boolean isInitial() {
```

```
        return initial;
    }

    public Collection getTransitions() {
        return transitions;
    }

    public void addTransition(Transition t) {
        transitions.add(t);
    }

    public void removeTransition(Transition t) {
        if (!deletingState)
            transitions.remove(t);
    }

    // this method will be called even before the constructor is called (because of
    // the super class constructor) so extra care must be taken with null pointers
    public void setBounds(int x, int y, int w, int h) {
        super.setBounds(x, y, w, h);

        if (circle != null)
            circle.setFrame(x+1, y+1, DIAMETER, DIAMETER);

        if (transitions != null) {
            Iterator i = transitions.iterator();
            while (i.hasNext())
                ((Transition)i.next()).update();
        }

        if (label != null)
            label.update();
    }

    public boolean contains(int x, int y) {
        return circle.contains(x, y);
    }

    public void edit() {
        StateEditor se = new StateEditor(null, this);
    }

    public void delete() {
        deletingState = true;

        // remove the state itself from the panel
        super.delete();

        // delete all transitions involving this state
        Iterator i = transitions.iterator();
        while (i.hasNext())
            ((Transition)i.next()).delete();

        transitions.clear();

        // delete the state label
        if (label != null)
            label.delete();


        drawingPanel.getProcess().removeState(this);

        deletingState = false;
    }

    public void draw(Graphics2D g2) {
        if (initial)
            g2.setPaint(currentColor.brighter());
```

```
        else
            g2.setPaint(currentColor);

        g2.fill(circle);

        g2.setStroke(STROKE);
        g2.setPaint(CONTOUR_COLOR);
        g2.draw(circle);
    }

    public void setDrawingPanel(DrawingPanel dp) {
        super.setDrawingPanel(dp);

        if (label != null)
            drawingPanel.addComponent(label, false);
    }

    public void writeXML(PrintWriter writer) {
        writer.println("<state id=\"" + id + "\">");

        if (center != null)
            writer.println("<point x=\"" + center.x + "\" y=\"" + center.y + "\"/>"
                );

        if (label != null)
            label.writeXML(writer);

        writer.println("</state>");
    }

    public void writeDivine(PrintWriter writer) {
        writer.print(getName());
    }
}
```

C.29 StateLabel

Listing C.29: Source code for StateLabel

```
package paxion;

import java.io.*;
import java.awt.*;

public class StateLabel extends AnchorableLabel {
    public StateLabel(String text) {
        super(text);
    }

    public StateLabel(String text, Point position) {
        super(text, position);
    }

    public void delete() {
        super.delete();

        ((State)parent).setLabel(null);
    }

    public void writeXML(PrintWriter writer) {
        writer.print("<label ");
        super.writeXML(writer);
        writer.println("</label>");
    }
```

```
}
```

C.30 Synchro

Listing C.30: Source code for Synchro

```
package paxion;

import java.io.*;
import java.awt.*;

public class Synchro extends AnchorableLabel {
    public Synchro(String text) {
        super(text);
    }

    public Synchro(String text, Point position) {
        super(text, position);
    }

    public void delete() {
        super.delete();

        ((Transition)parent).setSynchro(null);
    }

    public void writeXML(PrintWriter writer) {
        writer.print("<synchro ");
        super.writeXML(writer);
        writer.println("</synchro>");
    }

    public void writeDivine(PrintWriter writer) {
        writer.print("sync " + text + ";");
    }
}
```

C.31 TransitionEditor

Listing C.31: Source code for TransitionEditor

```
package paxion;

import java.awt.*;
import javax.swing.*;

public class TransitionEditor extends ObjectEditor {
    private JTextField tfGuard;
    private JTextField tfSynchro;
    private JTextField tfEffect;

    public TransitionEditor(Frame frame, Transition transition) {
        super(frame, "Edit Transition", transition);
    }

    protected JPanel createObjectPropertiesPanel() {
        JPanel panel;
        JLabel label;
```

```
        // set up the fields panel

        panel = new JPanel();
        panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
        panel.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));

        label = new JLabel("guard:");
        label.setAlignmentX(Component.CENTER_ALIGNMENT);
        tfGuard = new JTextField(((Transition)object).getGuardText());

        panel.add(label);
        panel.add(tfGuard);

        label = new JLabel("synchro:");
        label.setAlignmentX(Component.CENTER_ALIGNMENT);
        tfSynchro = new JTextField(((Transition)object).getSynchroText());

        panel.add(label);
        panel.add(tfSynchro);

        label = new JLabel("effect:");
        label.setAlignmentX(Component.CENTER_ALIGNMENT);
        tfEffect = new JTextField(((Transition)object).getEffectText());

        panel.add(label);
        panel.add(tfEffect);

        return panel;
    }

    protected void updateObject() {
        ((Transition)object).setGuardText(tfGuard.getText());
        ((Transition)object).setSynchroText(tfSynchro.getText());
        ((Transition)object).setEffectText(tfEffect.getText());
    }
}
```

C.32 Transition

Listing C.32: Source code for Transition

```
package paxion;

import java.io.*;
import java.util.*;
import java.awt.*;
import java.awt.geom.*;

public class Transition extends AbstractComponent {
    private State originState;
    private State destinationState;

    private Point originPoint = null;
    private Point destinationPoint = null;

    private Point guardAnchorPoint = new Point();
    private Point synchroAnchorPoint = new Point();
    private Point effectAnchorPoint = new Point();

    private Guard guard = null;
    private Synchro synchro = null;
    private Effect effect = null;

    private Collection intermediatePoints = new HashSet();
```

```
    private Arrow arrow;

    public Transition(State o, State d) {
        this(o, d, null);
    }

    public Transition(State o, State d, Arrow a) {
        originState = o;
        destinationState = d;

        if (a == null) {
            originPoint = new Point();
            destinationPoint = new Point();

            arrow = new Arrow(originPoint, destinationPoint);
        } else {
            arrow = a;

            // this is a bit of a hack. It "reuses" the arrow extremities points (
                even
            // though they will be updated shortly) so that the arrow is (
                graphically)
            // updated when those points are updated
            originPoint = arrow.getOrigin();
            destinationPoint = arrow.getDest();

            // create one TransitionPoint for each point in the arrow
            // we use a listIterator instead of a normal iterator to skip first and
                 last points
            Iterator it = arrow.getPoints().listIterator(1);
            for (int i = 1; i < arrow.getNumberPoints() - 1; i++)
                addArrowPoint((Point)it.next());
        }

        originState.addTransition(this);
        destinationState.addTransition(this);

        update();
    }

    public void addArrowPoint(Point p) {
        TransitionPoint ap = new TransitionPoint(this, p);
        intermediatePoints.add(ap);

        if (drawingPanel != null)
            drawingPanel.addComponent(ap, true);
    }

    public void insertPoint(Point p) {
        addArrowPoint(p);

        if (arrow != null)
            arrow.insertPointBeforeLast(p);

        update();
    }

    public void removePoint(Point p) {
        if (arrow != null)
            arrow.removePoint(p);

        update();
    }

    public void update() {
        updateAnchorPoints();
        arrow.update();
```

```
        setBounds(arrow.getBounds());

        if (guard != null)
            guard.update();

        if (synchro != null)
            synchro.update();

        if (effect != null)
            effect.update();
    }

    public void updateAnchorPoints() {
        // get current position of states

        Point o = originState.getCenter();
        Point d = destinationState.getCenter();

        Point p1, p2;
        double theta;

        int numberPoints = arrow.getNumberPoints();

        p1 = o;
        if (numberPoints > 2)
            p2 = arrow.getPoint(1);
        else
            p2 = d;

        theta = Math.atan2(p2.getY() - p1.getY(), p2.getX() - p1.getX());
        originPoint.setLocation(o.getX() + Math.cos(theta) * State.RADIUS, o.getY()
             + Math.sin(theta) * State.RADIUS);

        //----

        if (numberPoints > 2)
            p1 = arrow.getPoint(numberPoints - 2);
        else
            p1 = o;
        p2 = d;

        theta = Math.atan2(p2.getY() - p1.getY(), p2.getX() - p1.getX());
        destinationPoint.setLocation(d.getX() - Math.cos(theta) * State.RADIUS, d.
            getY() - Math.sin(theta) * State.RADIUS);

        //----

        theta = Math.atan2(d.getY() - o.getY(), d.getX() - o.getX());

        double dx = Math.cos(theta);
        double dy = Math.sin(theta);

        double distance = Point2D.distance(o.getX(), o.getY(), d.getX(), d.getY());

        guardAnchorPoint.setLocation(o.getX() + dx * distance/3, o.getY() + dy *
            distance/3);
        synchroAnchorPoint.setLocation(o.getX() + dx * distance/2, o.getY() + dy *
            distance/2);
        effectAnchorPoint.setLocation(o.getX() + dx * distance*2/3, o.getY() + dy *
             distance*2/3);
    }

    public State getOrigin() {
        return originState;
    }

    public State getDestination() {
        return destinationState;
```

```
    }

    public void setGuard(Guard guard) {
        this.guard = guard;
        if (guard != null) {
            guard.setParent(this);
            guard.setAnchorPoint(guardAnchorPoint);
            if (drawingPanel != null)
                drawingPanel.addLabel(guard);
        }
    }

    public void setSynchro(Synchro synchro) {
        this.synchro = synchro;
        if (synchro != null) {
            synchro.setParent(this);
            synchro.setAnchorPoint(synchroAnchorPoint);
            if (drawingPanel != null)
                drawingPanel.addLabel(synchro);
        }
    }

    public void setEffect(Effect effect) {
        this.effect = effect;
        if (effect != null) {
            effect.setParent(this);
            effect.setAnchorPoint(effectAnchorPoint);
            if (drawingPanel != null)
                drawingPanel.addLabel(effect);
        }
    }

    public AnchorableLabel getGuard() {
        return guard;
    }

    public AnchorableLabel getSynchro() {
        return synchro;
    }

    public AnchorableLabel getEffect() {
        return effect;
    }

    public void setGuardText(String text) {
        if (guard == null) {
            if (!text.equals(""))
                setGuard(new Guard(text, new Point(0, 0)));
        } else {
            if (text.equals(""))
                guard.delete();
            else
                guard.setText(text);
        }
    }

    public void setSynchroText(String text) {
        if (synchro == null) {
            if (!text.equals(""))
                setSynchro(new Synchro(text, new Point(0, 0)));
        } else {
            if (text.equals(""))
                synchro.delete();
            else
                synchro.setText(text);
        }
    }
```

```
public void setEffectText(String text) {
    if (effect == null) {
        if (!text.equals(""))
            setEffect(new Effect(text, new Point(0, 0)));
    } else {
        if (text.equals(""))
            effect.delete();
        else
            effect.setText(text);
    }
}

public String getGuardText() {
    if (guard != null)
        return guard.getText();
    else
        return "";
}

public String getSynchroText() {
    if (synchro != null)
        return synchro.getText();
    else
        return "";
}

public String getEffectText() {
    if (effect != null)
        return effect.getText();
    else
        return "";
}

public boolean contains(int x, int y) {
    return arrow.contains(x, y);
}

public void setSelected(boolean selected) {
    super.setSelected(selected);
    arrow.setColor(currentColor);
}

public void edit() {
    TransitionEditor te = new TransitionEditor(getFrame(), this);
}

public void delete() {
    // remove the state itself from the panel
    super.delete();

    // delete labels
    if (guard != null)
        guard.delete();

    if (synchro != null)
        synchro.delete();

    if (effect != null)
        effect.delete();

    // remove intermediate points from the drawingPanel
    Iterator i = intermediatePoints.iterator();
    while (i.hasNext())
        drawingPanel.removeComponent((TransitionPoint)i.next());

    drawingPanel.getProcess().removeTransition(this);

    arrow = null;
```

```
    }

    public void paintComponent(Graphics g) {
        arrow.draw((Graphics2D) g);
    }

    public void draw(Graphics2D g2) {
        arrow.draw(g2);
    }

    public void setDrawingPanel(DrawingPanel dp) {
        super.setDrawingPanel(dp);

        if (guard != null)
            drawingPanel.addComponent(guard, false);

        if (synchro != null)
            drawingPanel.addComponent(synchro, false);

        if (effect != null)
            drawingPanel.addComponent(effect, false);

        // add all intermediate points (TransitionPoint) to the drawingPanel
        Iterator i = intermediatePoints.iterator();
        while (i.hasNext())
            drawingPanel.addComponent((TransitionPoint)i.next(), true);
    }

    public void writeXML(PrintWriter writer) {
        writer.println("<transition from=\"" + originState.getId() + "\" to=\"" +
            destinationState.getId() + "\">");

        // we use a listIterator instead of a normal iterator to skip first and
            last points
        Iterator it = arrow.getPoints().listIterator(1);
        for (int i = 1; i < arrow.getNumberPoints() - 1; i++) {
            Point p = (Point)it.next();
            writer.println("<point x=\"" + p.x + "\" y=\"" + p.y + "\"/>");
        }


        if (guard != null)
            guard.writeXML(writer);

        if (synchro != null)
            synchro.writeXML(writer);

        if (effect != null)
            effect.writeXML(writer);

        writer.println("</transition>");
    }

    public void writeDivine(PrintWriter writer) {
        originState.writeDivine(writer);
        writer.print(" -> ");
        destinationState.writeDivine(writer);

        writer.print(" { ");

        if (guard != null)
            guard.writeDivine(writer);

        if (synchro != null)
            synchro.writeDivine(writer);

        if (effect != null)
            effect.writeDivine(writer);
```

```
        writer.print(" }");
    }
}
```

C.33 TransitionPoint

Listing C.33: Source code for TransitionPoint

```
package paxion;

import java.awt.*;
import javax.swing.*;

public class TransitionPoint extends CenteredComponent {
    // ---- public constants ----

    public static final int SIDE_LENGTH = 10;
    public static final Color DEFAULT_FILL_COLOR = new Color(0.3f, 0.5f, 0.9f);

    // ---- public static variables ----

    // "replaces" AbstractComponent's normalColor
    public static Color fillColor = DEFAULT_FILL_COLOR;

    // ---- private constants ----

    private static final Color CONTOUR_COLOR = Color.BLACK;
    private static final BasicStroke STROKE = new BasicStroke(1.0f);

    // ---- private variables ----

    private Rectangle square = new Rectangle(new Dimension(SIDE_LENGTH, SIDE_LENGTH
        ));
    private Transition transition = null;

    /**
     *
     */
    public TransitionPoint(Transition t, Point p) {
        super(p);

        setSize(SIDE_LENGTH+2, SIDE_LENGTH+2);
        setDeltaToCenter(SIDE_LENGTH/2 + 1, SIDE_LENGTH/2 + 1);

        currentColor = fillColor;

        transition = t;
    }

    public Color getNormalColor() {
        return fillColor;
    }

    public Transition getTransition() {
        return transition;
    }

    public void delete() {
        // remove the component from the panel
        super.delete();

        if (transition != null)
            transition.removePoint(center);
```

```
    }

    public void edit() {
    }

    public void setPosition(int x, int y) {
        super.setPosition(x, y);

        if (transition != null)
            transition.update();

        square.setLocation(x+1, y+1);
    }

    /* (non-Javadoc)
     * @see AbstractComponent#draw(java.awt.Graphics2D)
     */
    public void draw(Graphics2D g2) {
        g2.setStroke(STROKE);
        g2.setPaint(currentColor);
        g2.fill(square);
        g2.setPaint(CONTOUR_COLOR);
        g2.draw(square);
    }

    // we don't want editAction
    protected AbstractAction[] getContextMenuActions() {
        return new AbstractAction[] {deleteAction};
    }
}
```

C.34 Utils

Listing C.34: Source code for Utils

```
package paxion;

import java.io.File;
import java.net.URL;
import javax.swing.ImageIcon;

public class Utils {
    public static String getExtension(File f) {
        String ext = null;
        String s = f.getName();
        int i = s.lastIndexOf('.');

        if (i > 0 &&  i < s.length() - 1) {
            ext = s.substring(i+1).toLowerCase();
        }
        return ext;
    }

    // Load an icon (returns null if the path was invalid)
    public static ImageIcon loadIcon(String iconPath, boolean small) {
        if (iconPath == null)
            return null;

        if (small)
            iconPath += "16";
        else
            iconPath += "24";

        iconPath += ".gif";
```

```
        URL imageURL = ClassLoader.getSystemResource(iconPath);
        if (imageURL == null) {
            System.err.println("Resource not found: " + iconPath);
            return null;
        } else {
            return new ImageIcon(imageURL);
        }
    }

    /*
     * & - &amp;
     * < - &lt;
     * > - &gt;
     * " - &quot;
     * ' - &#39;
     */
    public static String convertInvalidXMLChars(String input) {
        final char badChars[] = new char[] {'&', '<', '>', '"', '\''};
        final String goodChars[] =
            new String[] {"&amp;", "&lt;", "&gt;", "&quot;", "&#39;"};

        String output = new String();

        for (int i=0; i<input.length(); i++) {
            boolean charReplaced = false;

            for (int j=0; j<badChars.length; j++)
                if (input.charAt(i) == badChars[j]) {
                    output += goodChars[j];
                    charReplaced = true;
                    continue;
                }

            if (!charReplaced)
                output += input.charAt(i);
        }
        return output;
    }
}
```

C.35 VariableEditor

Listing C.35: Source code for VariableEditor

```
package paxion;

import java.awt.*;
import javax.swing.*;

public class VariableEditor extends ObjectEditor {
    private JTextField tfName;
    private JComboBox cbType;
    private JSpinner spLength;
    private JTextField tfInitialValue;

    public VariableEditor(Frame frame, Variable variable) {
        super(frame, "Edit Variable", variable);
    }

    protected JPanel createObjectPropertiesPanel() {
        JPanel propertiesPanel;
        JPanel panel;
        JLabel label;
```

```
propertiesPanel = new JPanel();
propertiesPanel.setLayout(new BoxLayout(propertiesPanel, BoxLayout.Y_AXIS))
    ;
propertiesPanel.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));

panel = new JPanel();
panel.setLayout(new BoxLayout(panel, BoxLayout.LINE_AXIS));

label = new JLabel("type:");
panel.add(label);

panel.add(Box.createRigidArea(new Dimension(10, 0)));

cbType = new JComboBox(Variable.typeString);
cbType.setSelectedIndex(((Variable)object).getType());
panel.add(cbType);

propertiesPanel.add(panel);

propertiesPanel.add(Box.createRigidArea(new Dimension(0, 10)));

panel = new JPanel();
panel.setLayout(new BoxLayout(panel, BoxLayout.LINE_AXIS));

label = new JLabel("name:");
panel.add(label);

panel.add(Box.createRigidArea(new Dimension(10, 0)));

tfName = new JTextField(((Variable)object).getName());
tfName.setMaximumSize(new Dimension(Short.MAX_VALUE, Short.MAX_VALUE));
panel.add(tfName);

propertiesPanel.add(panel);

propertiesPanel.add(Box.createRigidArea(new Dimension(0, 10)));

panel = new JPanel();
panel.setLayout(new BoxLayout(panel, BoxLayout.LINE_AXIS));

label = new JLabel("array length:");
panel.add(label);

panel.add(Box.createRigidArea(new Dimension(10, 0)));

SpinnerModel lengthModel = new SpinnerNumberModel(((Variable)object).
    getLength(), 0, Integer.MAX_VALUE, 1); // initial value, min, max, step
spLength = new JSpinner(lengthModel);
panel.add(spLength);

propertiesPanel.add(panel);

propertiesPanel.add(Box.createRigidArea(new Dimension(0, 10)));

panel = new JPanel();
panel.setLayout(new BoxLayout(panel, BoxLayout.LINE_AXIS));

label = new JLabel("initial value:");
panel.add(label);

panel.add(Box.createRigidArea(new Dimension(10, 0)));

int initialValue = ((Variable)object).getInitialValue();
if (initialValue > -1)
    tfInitialValue = new JTextField(Integer.toString(initialValue));
else
    tfInitialValue = new JTextField();
```

```
        panel.add(tfInitialValue);

        propertiesPanel.add(panel);

        return propertiesPanel;
    }

    protected void updateObject() {
        ((Variable)object).setType(cbType.getSelectedIndex());
        ((Variable)object).setLength(((Integer)spLength.getValue()).intValue());

        if (!tfName.getText().equals(""))
            ((Variable)object).setName(tfName.getText());

        int initialValue;
        try {
            initialValue = Integer.parseInt(tfInitialValue.getText());
        } catch (NumberFormatException e) {
            initialValue = -1;
        }
        ((Variable)object).setInitialValue(initialValue);
    }
}
```

C.36 Variable

Listing C.36: Source code for Variable

```
package paxion;

import java.io.*;

public class Variable implements Editable {
    public static final String[] typeString = {"int", "byte"};
    public static final int numberTypes = typeString.length;

    private int type;

    // size of the array
    private int length;

    private int initialValue; //-1 = no initial value
    private String name;

    public Variable() {
        this(0, "", 0, -1);
    }

    public Variable(int type, String name) {
        this(type, name, 0, -1);
    }

    public Variable(int type, String name, int length, int initialValue) {
        this.type = type;
        this.name = name;
        this.length = length;
        this.initialValue = initialValue;
    }

    public void setType(int type) {
        this.type = type;
    }
```

```
public int getType() {
    return type;
}

public void setLength(int length) {
    this.length = length;
}

public int getLength() {
    return length;
}

public void setInitialValue(int initialValue) {
    this.initialValue = initialValue;
}

public int getInitialValue() {
    return initialValue;
}

public void setName(String name) {
    this.name = name;
}

public String getName() {
    return name;
}

public void edit() {
    VariableEditor ve = new VariableEditor(null, this);
}

public void writeXML(PrintWriter writer) {
    String s = "<variable type=\""+ type + "\" name=\"" + name + "\"";

    if (length > 0)
        s += " length=\"" + length + "\"";

    if (initialValue > -1)
        s += " initialValue=\"" + initialValue + "\"";

    s += "/>";

    writer.println(s);
}

public void writeDivine(PrintWriter writer) {
    writer.print(toDivineString());
}

public String toString() {
    if (name.equals(""))
        return "";
    else
        return typeString[type] + " " + toDivineString();
}

private String toDivineString() {
    if (name.equals("")) {
        System.err.println("Warning: Empty variable name detected !");
        return "";
    } else {
        String s = new String();

        s += name;

        if (length > 0)
            s += "[" + length + "]";
```

```
            if (initialValue > -1)
                s += " = " + initialValue;

            return s;
        }
    }
}
```

C.37 VectorListModel

Listing C.37: Source code for VectorListModel

```
package paxion;

import java.util.*;
import javax.swing.*;

public class VectorListModel extends AbstractListModel implements Collection {
    protected Vector data = new Vector();

    public int getSize() {
        return data.size();
    }

    public Object getElementAt(int index) {
        return data.elementAt(index);
    }

    //----

    // convenience method
    public void fireContentChanged(int index) {
        fireContentsChanged((Object)this, index, index);
    }

    //----

    public boolean add(Object o) {
        int index = data.size();
        if (data.add(o)) {
            fireIntervalAdded(this, index, index);
            return true;
        } else
            return false;
    }

    public boolean addAll(Collection c) {
        int index = data.size();
        if (data.addAll(c)) {
            fireIntervalAdded(this, index, data.size() - 1);
            return true;
        } else
            return false;
    }

    public void clear() {
        int size = data.size();
        data.clear();
        if (size > 0)
            fireIntervalRemoved(this, 0, size - 1);
    }

    public boolean contains(Object o) {
```

```
        return data.contains(o);
    }

    public boolean containsAll(Collection c) {
        return data.containsAll(c);
    }

    public boolean equals(Object o) {
        return data.equals(o);
    }

    public int hashCode() {
        return data.hashCode();
    }

    public boolean isEmpty() {
        return data.isEmpty();
    }

    public Iterator iterator() {
        return new VectorIterator();
    }

    public boolean remove(Object o) {
        int index = data.indexOf(o);
        if (index > -1) {
            data.remove(o);
            fireIntervalRemoved(this, index, index);
            return true;
        }
        return false;
    }

    public boolean removeAll(Collection c) {
        boolean changed = false;

        Iterator i = c.iterator();
        while (i.hasNext())
            if (remove(i.next()))
                changed = true;

        return changed;
    }

    public boolean retainAll(Collection c) {
        boolean changed = false;

        Iterator i = iterator();
        while (i.hasNext())
            if (!c.contains(i.next())) {
                i.remove();
                changed = true;
            }

        return changed;
    }

    public int size() {
        return data.size();
    }

    public Object[] toArray() {
        return data.toArray();
    }

    public Object[] toArray(Object[] a) {
        return data.toArray(a);
    }
```

```
    private class VectorIterator implements Iterator {
        int position = 0;

        public boolean hasNext() {
            return position != data.size();
        }

        public Object next() {
            try {
                return data.elementAt(position++);
            } catch(IndexOutOfBoundsException e) {
                throw new NoSuchElementException();
            }
        }

        public void remove() {
            throw new UnsupportedOperationException();
        }
    }
}
```

C.38 XMLFileFilter

Listing C.38: Source code for XMLFileFilter

```
package paxion;

import java.io.File;
import javax.swing.filechooser.*;

public class XMLFileFilter extends FileFilter {

    // Accept all XML files and directories
    public boolean accept(File f) {
        if (f.isDirectory())
            return true;

        String extension = Utils.getExtension(f);
        if (extension != null) {
            if (extension.equals("xml"))
                return true;
            else
                return false;
        }

        return false;
    }

    // The description of this filter
    public String getDescription() {
        return "XML Files (*.xml)";
    }
}
```